

The F-Basic System User's Manual

COPYRIGHT

The FastCom compiler for the F-Basic language is copyrighted (c) 1988 by Delphi Noetic Systems, Inc. The F-Basic User's Manual is also copyrighted (c) 1988 by Delphi Noetic Systems, Inc. All rights are reserved. This document may not, in whole or in part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without the prior written consent of Delphi Noetic Systems, Inc. The enclosed F-Basic Software License Agreement shall at all times govern the use of the manual and the software.

DISCLAIMER

The enclosed F-Basic Software License Agreement contains specific statements that provide you with a limited warranty concerning this manual, the software, and the physical media upon which it is stored. The agreement also limits the liability of Delphi Noetic Systems, Inc., and should be fully understood before this product is used.

TRADEMARKS

F-Basic is a registered trademark of Delphi Noetic Systems, Inc.
FastCom is a registered trademark of Delphi Noetic Systems, Inc.
Amiga is a registered trademark of Commodore-Amiga, Inc.
AmigaDOS is a registered trademark of Commodore-Amiga, Inc.
Amiga-Basic is a product of Microsoft Corporation
Snobol4 is a registered trademark of Bell Telephone Laboratories

ACKNOWLEDGEMENTS

Special thanks is due to Mr. Neal Countryman for his coordination of our beta-test sites, and to Mr. & Mrs. Richard Huxford and Ms. Terri Lemmon for their help in the preparation and design of this manual.

The F-Basic System User's Manual
Table of Contents

Chapter 1-Writing Programs In F-Basic And Using
The FastCom Compiler

Section 1-An Overview Of The F-Basic System.....	1.1
Section 2-Creating, Compiling, And Executing An F-Basic Program.....	1.3
Section 3-FastCom Compiler Options.....	1.5
Section 4-F-Basic Compiler Directives.....	1.8
Section 5-Compile Time Errors, Run Time Errors, And Debugging.....	1.8
Section 6-Using A RAM: Disk To Speed Program Development.....	1.10
Section 7-Technical Remarks Concerning Stand Alone Object Files.....	1.11
Section 8-Remarks On Using The F-Basic Manual.....	1.11

Chapter 2-F-Basic Program Structure

Section 1-Main Program Structure.....	2.1
---------------------------------------	-----

Chapter 3-Variables And Data In F-Basic

Introduction.....	3.1
Section 1-Integer Data.....	3.1
Section 2-Real Data.....	3.2
Section 3-Text Data.....	3.3
Section 4-More Technical Matters Concerning INTEGERS And REALS.....	3.5
Section 5-DATA Statements.....	3.6
Section 6-CONSTANT Declarations.....	3.7

Chapter 4-Operators, Expressions, & Assignments
In F-Basic

Introduction.....	4.1
Section 1-Binary Arithmetic On INTEGERS And REALS.....	4.1
Section 2-Relational Operators On INTEGER, REAL And TEXT.....	4.3
Section 3-Logical Operators.....	4.4
Section 4-The MAX And MIN Operators.....	4.5
Section 5-Shift Operators On INTEGERS.....	4.5

Section 6-String Concatenation.....	4.6
Section 7-Unary Arithmetic And Unary Logical Operators.....	4.6
Section 8-PEEK, POKE, And Address Operators.....	4.6
Section 9-Operator Precedence And Expressions.....	4.8
Section 10-Assignment Statements And The Immediate Assignment Operator.....	4.10

Chapter 5-Control Structures In F-Basic

Introduction.....	5.1
Section 1-Statement Labels, Statement Numbers, And The GOTO Statement.....	5.1
Section 2-GOSUBS And Local Subprograms.....	5.3
Section 3-IF Statements.....	5.4
Section 4-The WHEN Statement (Case Structures).....	5.6
Section 5-FOR Loops And The EXIT Statement.....	5.8
Section 6-The WHILE Statement.....	5.10
Section 7-The REPEAT Statement.....	5.11
Section 8-The ON...GOTO And ON...GOSUB Statements..	5.12
Section 9-Conclusion.....	5.13

Chapter 6-Keyboard Input And Screen Output In F-Basic

Section 1-Keyboard Input.....	6.1
Section 2-Unformatted Screen Output.....	6.3
Section 3-Formatted Screen Output.....	6.4
Section 4-Line And Blank Control Using PRINT.....	6.6
Section 5-Cursor Control In F-Basic.....	6.7

Chapter 7-File Input And Output In F-Basic

Introduction.....	7.1
Section 1-Opening A File.....	7.1
Section 2-Inputting Data From A File.....	7.3
Section 3-Printing To A File.....	7.4
Section 4-Skipping To The End Of A File.....	7.5
Section 5-Closing A File.....	7.6

Chapter 8-Arrays In F-Basic

Section 1-One Dimensional Arrays.....	8.1
Section 2-Array Equates.....	8.2
Section 3-DATA Statements And Arrays.....	8.2
Section 4-The IN Operator And The SETARRAY Statement.....	8.4
Section 5-Multi-Dimensional Arrays In F-Basic.....	8.5

Chapter 9-Subprograms In F-Basic

Introduction.....	9.1
Section 1-Overall Program Structure.....	9.1
Section 2-Passing Information Between Program Units.....	9.2
Section 3-GLOBAL Variables And The Declaration Part Of The Main Program.....	9.2
Section 4-Parameters And The Declaration Part Of Subprograms.....	9.4
Section 5-Invoking Subprograms And Passing Parameters.....	9.5
Section 6-The Executable Part Of A Subprogram.....	9.7
Section 7-A More Detailed Discussion Of Passing Information Via Parameters.....	9.8
Section 8-Recursion In F-Basic.....	9.9

Chapter 10-Using INCLUDE And APPEND To Combine Program Files

Introduction.....	10.1
Section 1-The INCLUDE Statement.....	10.1
Section 2-The APPEND Statement.....	10.2

Chapter 11-Programming The Amiga's Sound Facilities

Introduction.....	11.1
Section 1-TRANSLATE, NARRATE, And VOICE.....	11.1
Section 2-TONE.....	11.3
Section 3-WAVE, SOUND, And WAVE_END.....	11.4

Chapter 12-High Level Graphics In F-Basic

Introduction.....	12.1
Section 1-Custom Screens In F-Basic.....	12.1
Section 2-Windows In F-Basic.....	12.4
Section 3-Text In F-Basic Windows.....	12.8
Section 4-Colors, Points, Lines, And Drawing In F-Basic Windows.....	12.9
Section 5-Patterned Graphics In F-Basic.....	12.13
Section 6-Manipulating Image Blocks In F-Basic.....	12.16
Section 7-Menus In F-Basic.....	12.20

Chapter 13-Event Trapping In F-Basic

Introduction.....	13.1
Section 1-Events In F-Basic.....	13.1
Section 2-Activating Events And Enabling Event	

Checking.....	13.2
Section 3-More About F-Basic Events And Their Associated System Variables.....	13.4

Chapter 14-RECORD Structures In F-Basic

Introduction.....	14.1
Section 1-Creating A RECORD Type.....	14.1
Section 2-Declaring And Referencing RECORD Variables.....	14.3
Section 3-Using RECORD References In Assignments And Expressions.....	14.4
Section 4-Pointers In F-Basic.....	14.5
Section 5-Assigning Values To Pointers And Dynamic Memory Management.....	14.6
Section 6-DATA Statements Involving RECORD Variables.....	14.8
Section 7-More On RECORDs And Pointers.....	14.9

Chapter 15-F-Basic Library Functions And Macros

Introduction.....	15.1
Section 1-Operations On Integers.....	15.1
Section 2-Operations On Real Numbers.....	15.5
Section 3-Operations Involving Text Expressions....	15.7
Section 4-Functions That Convert One Data Type To Another.....	15.9
Section 5-Functions That Interact With AmigaDOS....	15.11

Chapter 16-Accessing Amiga ROM Libraries From F-Basic

Introduction.....	16.1
Section 1-A Summary Of Available Amiga Libraries...	16.1
Section 2-F-Basic Access To An Amiga Library Routine.....	16.1
Section 3-An Example Program.....	16.3

Chapter 17-Interacting Directly With The 68000 Processor

Section 1-Accessing Processor Registers.....	17.1
Section 2-Saving And Restoring Processor Register Contents.....	17.2
Section 3-Inserting Machine Code Within F-Basic Programs.....	17.3
Section 4-Executing Direct Machine Code.....	17.3

Chapter 18-Suggestions For More Efficient Use Of The F-Basic System

Introduction.....	18.1
Section 1-Optimized Versus Unoptimized Variable Storage.....	18.1
Section 2-Arrays And Pointers.....	18.2
Section 3-Speeding Up Integer Multiplies And Divides.....	18.3
Section 4-Auto Increment And Auto Decrement In Indirect Reference.....	18.3

Chapter 19-Pattern Matching In F-Basic

Introduction.....	19.1
Section 1-Patterns.....	19.1
Section 2-Pattern Matching.....	19.4
Section 3-Conditional And Immediate Assignment Operators.....	19.6
Section 4-Anchored And Unanchored Pattern Matching.....	19.8
Section 5-System Patterns.....	19.8

Appendix I-The ASCII Character Set

Appendix II-The Default Colors Of The 32 Amiga Color Registers

Appendix III-Precedence Tables For F-Basic Operators

Appendix IV-Index Of F-Basic Keywords, Functions, And Commands

Appendix V-FastCom Compile Time Error Messages

Appendix VI-F-Basic Run Time Error Messages

Appendix VII-Ordering Customized Versions Of The FastCom Compiler

Appendix VIII-Registering Commercial Stand Alone F-Basic Object Files

Writing Programs In F-Basic And Using The FastCom Compiler

Introduction

In this chapter, an overview of the F-Basic system is presented. This is followed by a step-by-step example of using the FastCom compiler to compile and then execute a simple program written in F-Basic. The chapter concludes with comments on the most effective use of the compiler and of this manual.

Section 1-An Overview Of The F-Basic System

Although this manual's presentation is reasonably complete, it is not intended to be a comprehensive textbook on computers or on computer programming. Familiarity with an Amiga programming language, such as Amiga Basic, is very helpful in learning to use the F-Basic system.

F-Basic is an enhanced extension of the BASIC programming language. BASIC is one of the older and more popular languages. It is a simple and powerful language that has been implemented on a wide variety of systems in its twenty or so years of existence. Most implementations of BASIC have certain disadvantages which have prompted many programmers to turn to more recently developed 'high level' languages. One of the main design goals in the creation of F-Basic was to retain the appealing simplicity and ease of use inherent in traditional BASIC while overcoming most of the objections associated with its use.

Most BASIC packages are implemented using an 'interpreter'. The F-Basic language, in contrast, employs a 'compiler'. An interpreter reads a user program one line at a time, decides what actions are required to execute that line, and then performs those actions. A compiler, on the other hand, reads a user program one line at a time, decides what actions are required to execute that line, and generates appropriate machine level code. The machine code for each line is collected together into a single file called the 'object file'. The object file can later be executed any number of times without the intervention of the compiler.

A simple analogy will help clarify the difference. Suppose you wish to read a novel written in Russian. You might hire a Russian interpreter to read the novel to you one line at a time, translating each line from Russian into English. Alternatively, you might ask the individual to translate the novel into English in book form. You could then read the novel at your convenience without the assistance of the interpreter. In fact, others could then read the novel at their leisure.

One obvious disadvantage of the interpretive process is speed. When an interpreter executes a program, it must go through the three step process outlined above. When executing a compiled program, on the other hand, only the third step is involved. The factor by which a program can be speeded up by compilation depends, of course, on the nature of the program, but can range from a value of four to one hundred. Programs which do a great deal of input/output or file accessing will exhibit the least amount of speed benefit from compilation, while those that are computationally intensive will exhibit the greatest benefit.

When developing a program, however, interpreters have an advantage over many compilers. Compilers frequently require an involved multi-step process of compilation, linking, and loading before the program can be executed. If the program contains logic errors, one must suffer through the tedious delays involved in this process at each cycle of program testing. F-Basic, on the other hand, overcomes these delays by compiling the user program at a rate comparable to interpreters, and by generating object files that can be directly executed without time consuming intermediate steps. Thus, F-Basic provides an ease of development comparable to interpreters, while achieving a much faster program execution speed.

A second group of objections to traditional BASIC stem from its failure to incorporate many of the sophisticated features present in modern day languages like Pascal or C. Capabilities such as local subprograms, recursion, global variables, record structures, extended string handling facilities, and a wider variety of control structures have been incorporated into F-Basic. This provides the F-Basic programmer with the same facilities present in the other modern languages.

One criticism of many of the compiled languages available for the Amiga is that they do not provide the programmer with a simplified high level access to the machine's graphics, sound, and speech facilities. Special care has been taken to incorporate in F-Basic a powerful and simple procedure for utilizing the Amiga's full capabilities. In addition, F-Basic provides direct access to the low level ROM Kernel routines for those programmers already comfortable with directly interacting with the Amiga's operating system.

Another area in which F-Basic differs significantly from most other Amiga languages is the speed of its floating point arithmetic. Computationally intensive programs, especially those involving real arithmetic, have been an area in which personal computers have often exhibited poor performance. The floating point arithmetic routines utilized by F-Basic programs have been designed to provide simultaneously more accuracy than standard single precision floating point arithmetic and a noticeably faster execution speed.

Finally, although F-Basic was designed as a single pass compiler to facilitate speed of compilation, it performs a number of local optimizations. For example, all logical AND and OR operations are computed in a 'short circuit' mode. Also, where possible, processor register storage is utilized for some INTEGER and REAL variables. As an illustration of the resulting performance, the FastCom compiler is itself written in F-Basic !

Section 2-Creating, Compiling, And Executing An F-Basic Program

The first step in developing an F-Basic program is to use an editor to create a source code file. One may choose any editor that can produce an ASCII file.

The line editor ED, bundled with your original F-Basic system disk, produces this type of file exclusively. To invoke ED, one types:

```
1> ED <source file name>
```

Here, <source file name> represents the name of the source file. For example, one might type:

```
1> ED FBasic.Example
```

to use the editor to create a source file named FBasic.Example. A complete description of the features of the simple line editor ED can be found in the reference 'AmigaDOS User's Manual'.

Alternatively, one may use the Amiga Basic editor to produce this type of file. After entering the source program into the system, one should save the file using the ,A extension of the SAVE command. This directs the Amiga Basic system to store the file in an ASCII format rather than its normal compressed format. Referring to the above example, one might type:

```
SAVE "FBasic.Example",A
```


in the Output Window in order to again produce a source file in an ASCII format.

A source file on the F-Basic system disk named FBasic.Example is provided in order to illustrate the remaining steps of program development. To examine its contents, one may simply type:

```
1> TYPE FBasic.Example
```

The second step in using the F-Basic system is to compile the source code file into an object code file. In the simplest case, the source code file is in the main directory of the system disk. The command:

```
1> FB FBasic.Example
```

will cause the FastCom compiler residing in file FB to be loaded into memory. The FastCom compiler then reads the source file FBasic.Example and produces an object file containing the 68000 machine code equivalent of the source file. This object file will be automatically named FBasic.Example.bin. By default, the object code file name is obtained by concatenating '.bin' to the source code file name.

The object file may be executed after compilation by typing:

```
1> FBasic.Example.bin
```

The AmigaDOS operating system then loads and runs the object file named FBasic.Example.bin. The reader may enter the above two commands to compile and execute the simple source file name FBasic.Example on the F-Basic system disk. The program prints and 'says' a simple greeting.

F-Basic program development, then, consists of three basic steps:

1. Create an ASCII source file using your favorite text editor.
2. Type 'FB <source file name>' to invoke the FastCom compiler and produce an object file name <source file name>.bin.
3. Type '<source file name>.bin' to execute the object file program.

The source file may reside in the main directory, or a subdirectory of the F-Basic system disk, or alternatively, on some other disk. When the source program is not part of the main directory of the F-Basic system disk, it is necessary to specify a complete AmigaDOS pathname as part of the <source file name>. For example:

```
1> FB Subdirect/FBasic.Example
```


would allow the FastCom compiler to access the source file FBasic.Example located in the subdirectory named 'Subdirect', and produce an object file name FBasic.Example.bin located in the same subdirectory. If the source file was resident on the first external drive, one would type:

```
1> FB dfl:FBasic.Example
```

to allow the FastCom compiler to produce an object file name FBasic.Example.bin on the disk in the first external disk drive.

Section 3-FastCom Compiler Options

The F-Basic system is flexible, and permits the user to vary the object file name as well as certain parameters in the FastCom compiler that extend the system capabilities. F-Basic programs of modest complexity do not require these options and the user may wish to omit this section on a first reading.

If the source file has the distinguished name 'Test', then one may invoke the compiler simply by typing 'FB', without a trailing <source file name>. This will naturally produce the object file name Test.bin.

The user may override the rule for producing object file names by supplying an object file name of his/her own choosing. Thus, typing:

```
1> FB <source file name> <object file name>
```

will cause the FastCom compiler to read a source file named <source file name> and produce an object file named <object file name>. For example, the command:

```
1> FB FBasic.Example FBasic.Nuts
```

will produce an object file version of FBasic.Example named FBasic.Nuts.

One may vary certain of the characteristics of the FastCom compiler by appending the keyword 'OPT' followed by any string composed of {-C,-G,-O,-P,-S} to the command that invokes the compiler. Examples of such commands are:

```
1> FB FBasic.Example opt-P
or 1> FB FBasic.Example opt-o100000-p-g
or 1> FB FBasic.Example FBasic.Nuts OPT-C5000-S5000
```


The effect and purpose of these compiler options is presented next.

<u>Compiler Option</u>	<u>Description</u>
-C<number of bytes>	This compiler option alters the maximum length in bytes of a buffer used in string concatenation. Its default size is 1000 bytes (characters). If the user program concatenates two strings whose combined length exceeds 1000 bytes, this option must be invoked with <number of bytes> chosen to be greater than the combined length of any string concatenation in the user program.
-G	Normally, the FastCom compiler saves the object file on disk from which it may be later loaded and executed by typing the object file name. During the process of debugging, the user may short circuit this moderately time consuming step by specifying the '-g' (GO) option. When this option is specified, the object file is not saved on disk but remains in memory and is immediately executed upon completion of compilation. Upon termination of each execution of the user program, the F-Basic system prompts the user to allow repeated execution of the object code for testing purposes.
-O<number of bytes>	The size of an object file is determined by the amount of data storage specified by the program and by the number of executable statements contained within the program. This compiler option alters the maximum length of an object file. The default value is 50,000 bytes. To accommodate larger object files, invoke this option with <number of bytes> chosen greater than the required object file size.
-P	This compiler option causes the FastCom compiler to print each line of the F-Basic source file to the screen as it

is being compiled. Although this slows compilation speed, it allows the user to easily locate the position of the first detected syntax error.

-S<number of bytes>

Literal character strings that occur in a user program are stored in a buffer during compile time. The default length of this buffer is 1000 bytes per program module. If a program generates an error indicating that this buffer length has been exceeded during compilation, then this option should be specified with <number of bytes> sufficiently large enough to accommodate the number of literal characters contained in the program.

Thus, the command:

```
1> FB FBasic.Example opt-P
```

will cause the FastCom compiler to read the source file FBasic.Example, print each line to the screen during compilation, and produce the object file FBasic.Example.bin.

The command:

```
1> FB FBasic.Example opt-o100000-p-g
```

will cause the FastCom compiler to read the source file FBasic.Example, print each line to the screen during compilation, allow the object file to be up to 100,000 bytes in length, and execute the object code instructions immediately upon the termination of compilation. NOTE: The object file FBasic.Example.bin will not be saved to the disk because the -g option was specified.

Finally, the command:

```
1> FB FBasic.Example FBasic.Nuts OPT-C5000-S5000
```

will cause the FastCom compiler to read the source file FBasic.Example, produce an object file named FBasic.Nuts, and allow a concatenated string length of up to 5000 characters as well as a total of 5000 characters of literal text string per module (main program or subprograms).

Section 4 - F-Basic Compiler Directives

Compiler directives are similar in one respect to compiler options, in that these directives also modify the characteristics and behavior of the FastCom compiler. They differ from compiler options, however, in that they occur within the source file itself, rather than at the end of the command line invoking the FastCom compiler.

The syntax of a compiler directive is:

&<directive>

Generally, compiler directives may occur at any line in the source program's executable part. The term 'executable part' is fully defined in the chapter entitled 'F-Basic Program Structure'. A complete list of compiler directives together with a brief description of each is presented in Appendix IV. Two examples of compiler directives are discussed in the next section.

Section 5-Compile Time Errors, Run Time Errors, And Debugging

Mistakes in F-Basic programs may be divided into two categories, syntax errors that are detected during compilation of the source file, and run time errors that occur during the execution of an object file. The term syntax refers to the body of rules that govern the construction of 'sentences' in a language. All languages, whether natural languages such as English or Russian, or computer languages such as F-Basic, Pascal, and C, have rules that permit some statements while regarding other statements as ungrammatical. The remainder of this manual is principally devoted to the syntax rules of F-Basic programs.

When a syntax error is encountered by the FastCom compiler, compilation ceases and an error message is generated. The message consists of a copy of the source line containing the error, an identification of the module in which the error is located, and a description of the syntax error involved. Appendix V contains a listing of the over 125 compile time error messages generated by the FastCom compiler. When such an error is encountered, it is necessary to return to the editor and modify the indicated line of the source file in order to correct the syntax.

This process may be facilitated by inserting the compiler directive:

`&EDITOR <AmigaDOS system command>`

in the executable part of the source program. For example, if the source file named `FBasic.Example` contained the compiler directive:

`&EDITOR ED FBasic.Example`

then anytime a syntax error in the source file was detected the appropriate error message as explained above would be generated. Additionally, after a delay of a few seconds to enable the user to study the error, control would pass to the line editor `ED` due to the contents of the string following the `&EDITOR` directive. In this example, `ED` would automatically load the source file `FBasic.Example`, making it extremely easy to locate and correct the offending line. After the correction has been made, the user should exit the editor and reinvoke the `FastCom` compiler. The use of the `&EDITOR` compiler directive allows the user to choose his/her favorite editor when working with the `F-Basic` system, rather than being constrained to the use of an unfamiliar editor.

There is both an up side and a down side to the strategy of detecting errors one line at a time. Some compilers continue to compile and diagnose successive syntax errors after encountering the first such error. All too often, the succeeding 'errors' may not really be violations of the rules of syntax, but result from confusion created by the first syntax error. A second disadvantage of this alternative strategy is that if a large number of errors are present, or if the source file is lengthy, there will be a substantial delay before the user may attempt to correct the first detected error. The developers of `F-Basic` have chosen the 'one error at a time' approach because they felt that this strategy makes the identification of syntax errors easier, prevents 'false' errors due to compiler confusion, and eliminates lengthy delays in compilation before error correction can be initiated. Because of the speed of the `FastCom` compiler, repeated compilations necessary to remove multiple syntax errors do not introduce excessive delays.

As mentioned above, run time errors are those that occur during the execution of the object file. A complete list of the run time errors generated by the `F-Basic` system is presented in Appendix VI. These errors do not result from violations of the rules of the language, but from mistakes in program logic. When a run time error is encountered, the `F-Basic` system displays an appropriate message and attempts to deallocate resources and gracefully return control to the `AmigaDOS` operating system. If the run time error is quite serious, however, parts of the `F-Basic` system or the `AmigaDOS` operating system may have been overwritten by the actions of the user program. In this case, the system must be rebooted. To protect against extreme errors in these circumstances, the user should always use a copy of the

F-Basic system disk during development.

In order to simplify debugging, it is frequently useful to inactivate sections of the user program in order to make the testing process easier. This procedure is conveniently accomplished by using the compiler directive:

&IGNORE 1

All lines of the source program encountered after this directive is issued will be treated as comments, and not compiled into object code. When the directive:

&IGNORE 0

is then encountered, the compilation process is re-initiated.

Section 6-Using A RAM: Disk To Speed Program Development

The term RAM: disk refers to assigning a block of the Amiga's random access memory (RAM) to simulate a disk file. Operations, then, which would normally require a disk access are converted into a memory access. Accessing memory may be over 100 times faster than disk accesses.

The sequence of commands necessary to place the F-Basic system into the Amiga's RAM: disk and then compile the sample program FBasic.Example from this new location will now be illustrated. This technique is strongly recommended for improved system response provided that sufficient memory is available (extended memory Amiga environments).

```
1> COPY FB TO RAM:
1> COPY FastLib TO RAM:
1> COPY FastError TO RAM:
1> COPY FBasic.Example TO RAM:
1> CD RAM:
1> FB FBasic.Example
```

When this technique is utilized, the object file will reside in the RAM: disk. When the development process has been completed, the object file should be copied back to a physical (floppy or hard) disk for permanent storage. If the user program includes the &SYSLIB directive to reference ROM Kernel routines, the file FastSysLib should also be copied from the F-Basic system disk to the RAM: disk before invoking the compiler. For a complete discussion of the &SYSLIB directive, see the chapter entitled 'Accessing Amiga ROM Libraries

From F-Basic'.

Section 7-Technical Remarks Concerning Stand Alone Object Files

A user program which can be executed without the presence of an interpreter or compiler is referred to as a 'stand alone' application. Most interpreters cannot support stand alone applications, because the interpreter is required to execute the program. Many compilers require that object libraries and other support files be linked with the user application in order to achieve stand alone capability. This can be a lengthy process.

Each object file produced by FastCom is one that will be accepted by the Amiga's operating system for direct execution. This makes the production of an F-Basic stand alone application extremely convenient. The process requires just two simple steps. First, the object file associated with the user application should be copied to a formatted user application disk. Finally, the FastLib accessory file must be copied to the application disk. Only one copy of FastLib need be present on a disk, regardless of the number of resident application programs.

For the policies concerning the commercial distribution of an F-Basic application, consult the appendix entitled 'Registering Commercial Stand Alone F-Basic Object Files'.

Section 8-Remarks On Using The F-Basic Manual

For beginning programmers, the number of features in the F-Basic language and the resultant size of the F-Basic manual might at first appear intimidating. This, however, need not be the case. The presentation of the F-Basic language has been organized in this manual proceeding from elementary and frequently used features to more complex or advanced capabilities. Thus, the reader just beginning programming will be able to construct routine programs based on a mastery of the first few chapters of this manual. This is especially true, as the material presented in the early chapters describes a language which differs only slightly from standard BASIC.

Secondly, most language features are illustrated with programs or program segments in the manual, and the F-Basic package includes, along with the system disk, a sample program disk containing a large

number of programs varying both in complexity and function. Special care has been taken to minimize 'forward references' within the manual. Thus, it is rarely necessary to skip ahead in the manual in order to understand a topic currently under discussion.

Once the user obtains some experience with the language, he/she may quickly reference an F-Basic feature by consulting the appendix entitled 'Index Of F-Basic Keywords, Functions, And Commands'. Here, an alphabetized list of the principle features, together with a cross-reference for a detailed discussion of each, is presented.

The F-Basic system comes with an extensive collection of library routines. These may be referenced in one of two ways. A discussion organized on the basis of the routine's purpose is presented in the chapter entitled 'F-Basic Library Functions And Macros'. An alphabetical reference to these routines is also present as a part of the above mentioned appendix.

F-Basic Program Structure

Section 1-Main Program Structure

Programs in F-Basic begin with the keyword PROGRAM. The statements following the PROGRAM statement are divided into two groups, the declaration part and the executable part. The program is terminated with the keyword END.

```
PROGRAM <name>

<declaration part>

<executable part>

END
```

The declaration part consists of type declarations, DATA statements, and CONSTANT declarations. These will be discussed in the next chapter. The common characteristic of these statements is that they generate no executable code.

The executable part consists of computations, data input and output, and F-Basic high level control statements. These types of program lines will be the primary subject of the remainder of this manual.

One major difference between F-Basic and some other language systems concerns the use of line numbers within a program. In F-Basic, lines should be numbered or labeled only if they are referenced by other lines in the program (typically to pass program control). Otherwise, line numbering of the program is unnecessary.

F-Basic allows the use of subprograms of both type SUBROUTINE and FUNCTION. These extensions to the basic program structure shown above are discussed in the chapter entitled 'Subprograms In F-Basic'.

Comment statements in F-Basic take two acceptable forms. A line beginning with the keyword 'REM' or with the symbol ? is taken as a comment statement and ignored by the compiler. As an example:

```
PROGRAM TEST
INTEGER I,J
REM This is a comment
? So is this
```


END

One may place several F-Basic statements on a single program line, separated by a semicolon. For example:

```
PROGRAM TEST
INTEGER I,J
I=2 ; J=5 ; ? Comment-I initialized to 2 and J to 5
PRINT I,J
END
```

Each program line in F-Basic is limited to a maximum length of 120 characters. Statements may not be continued from one program line to the next. The TAB character may be used anywhere within a line.

Variables And Data In F-Basic

Introduction

The purpose of most programs is to manipulate data. F-Basic supports three simple types of data:

1. INTEGER
2. REAL
3. TEXT

Data may be referred to 'literally', as in 123, 3.1415, or "This is a Text String". Such an explicit reference to data will be called a literal. A second way of referencing data is 'symbolically', as in NUMBER, PI, STRINGVAR. This form of data reference is called a variable name. In F-Basic a variable name must begin with a letter of the alphabet (or the underline symbol), contain only letters, digits, and the underline symbol, and contain not more than 20 characters. Blanks and special characters are not permitted in a variable name. For instance, NUM1, Day_of_Week, and File160 are valid in F-Basic. However, 6Days, Name.One, and UnitedStatesOfAmerica are invalid-each for one of these reasons. F-Basic distinguishes between upper and lower case letters; thus, NAME and Name are distinct variable names.

F-Basic requires that the program specify the type of data that is to be referenced by a variable name. For example, the commands:

```
INTEGER I,J,K
REAL X,Y
TEXT*30 String,NAMELIST
```

assign the first three variable names to INTEGER data, the next two variable names to REAL data, and the last two names to TEXT or strings of character data. Each of these three data types, with their corresponding declaration statements are the subject of this chapter.

Section 1-Integer Data

Variables whose values are whole numbers such as 12, -7, 0, or 10000, are declared using the INTEGER statement. The syntax of this declaration is:

INTEGER <list of variable names separated by commas>

For example:

INTEGER Time,PRICE,A(10)

establishes Time and PRICE as variables representing a single INTEGER. The variable A is called an 'array' and is discussed in the chapter entitled 'Arrays In F-Basic'. It represents a list of 10 integers. An INTEGER in F-Basic is a positive or negative whole number whose magnitude is less than 2,147,483,647 which is also 2 to the thirty-first power minus 1. For those desiring what are termed 'byte' and 'word' length integers, F-Basic provides those data types too. They are discussed near the end of this chapter.

Commas in F-Basic are used as separators. Thus, the expression 1,400 does not represent the integer 1400. Instead, it is a list of two numbers, 1 followed by 400.

Integer literals in F-Basic are by default decimal or base 10. In many applications, it is convenient to represent integers in other bases. For instance, many operating system interactions require the use of binary (base 2), octal (base 8), or hexadecimal (base 16) integer constants. F-Basic supports literal integers expressed relative to any base B where B ranges from 2 to 36. The digits to be employed are: 0,1,2,3,4,5,6,7,8,9,A,B,C,...,Y,Z. The symbol A is a 'digit' whose value is 10, while B has value 11, and continuing to Z which has value 35. The syntax of such a number is:

<string of digits whose value is less than the base> ' <integer base>

The <integer base> must be expressed as a decimal integer. For example, 22'5 is an integer base 5 whose value is 12 in ordinary decimal or base 10, while A5'16 is an integer base 16 whose base 10 value is 165. A further example is 120'36 whose value is 2556 base 10.

Section 2-Real Data

REAL numbers are decimal numbers or those with a fractional part. They may be represented in F-Basic in two different forms. 'Ordinary decimal notation' or 'fixed point' refers to representations like 12.10 or -32.1546. For representing very small or very large numbers, 'scientific notation' is supported by the F-Basic system. For example: 12.032(E15) , -3.1415(E-06) , and 1.234(E-123) are equivalent

respectively to 120320000000000000., -0.0000031415, 0.<122 zeros>1234.
The general syntax for scientific notation in F-Basic is:

<signed decimal number>(E<a signed integer of 3 or fewer digits>)

The value of such an expression is obtained by multiplying the decimal number by 10 raised to the power <a signed integer of 3 or fewer digits>. REAL numbers in F-Basic provide 9 significant digits of accuracy. Most single precision REAL numbers in other BASIC packages provide only six significant digits.

REAL variables in F-Basic are declared with the REAL statement, whose syntax is:

REAL <list of variables separated by commas>

For example:

```
REAL MyNum,B(3)
```

establishes MyNum as a single REAL variable, and B as an array or list of three REAL numbers.

Section 3-Text Data

A TEXT string, or 'string' for short, refers to a sequence of ASCII characters. A list of valid ASCII characters is provided in an Appendix to this manual. For example, "My Name Is" and "Jack Robensen" are literal TEXT strings. In general, a text literal is any sequence of ASCII characters enclosed in double quotes (""). On occasion, one may wish to change the double quote symbol used to bracket text literals to some other character. For instance, this would be necessary if one of the characters embedded within the literal was itself a double quote. The symbol used to bracket text literals at any given time is referred to as the 'delimiter'. The compiler directive &DELIMITER <single character> may be used at any point in the source program when it is desired to alter the current value of the delimiter. For example, the statements:

```
&DELIMITER !  
PRINT !This is now a text literal!
```

establish the current value of the delimiter as the ! symbol for the balance of the program, or until another &DELIMITER directive statement in the source program changes the delimiter character.

TEXT variables in F-Basic are declared using a TEXT statement, whose syntax is:

TEXT <list of text variables with optional length specification separated by commas>

or TEXT* <default length> <list of text variables with optional length specification separated by commas>

For instance:

```
TEXT Str1,Str2*5
TEXT*20 STR2,STR3*10
```

establishes 4 TEXT variables for use in the program. There is one major difference between strings in F-Basic and strings in most other BASIC packages. In F-Basic, a TEXT variable must be assigned a maximum length. This is done by appending the star or asterisk character and a positive integer to the end of the variable name. This integer represents the maximum length of the string to be stored in the variable. If the length specification is omitted, the variable is assigned a default length. The default length is determined by the length specification on the keyword TEXT (if present) as shown in the second example program line above. If a length specification is not provided on either the variable name or the keyword TEXT, then the system assigns the variable a length of 120 characters. Thus, in the above examples, Str1 is assigned a length of 120, Str2 a length of 5, STR2 a length of 20, and STR3 a length of 10.

Most implementations of BASIC reference a 'substring' of a string variable by using the function MID\$. F-Basic provides a more compact and general purpose way of referencing substrings. The syntax is:

<TEXT variable name> (<first character of substring>:<last character of substring>)

Here, <first character of substring> and <last character of substring> are integer expressions giving the position of the first character and the last character of the substring counting from the left end of the string. For example:

```
PROGRAM SubString
TEXT STR*50
STR="This Is A Substring Example"
PRINT STR(11:11+8)
END
```

will display the characters 'Substring' on the screen. If the program

contained the line `PRINT STR(1:4)` instead, the characters 'This' would be displayed. As a special case, a single character of the substring may be referenced by:

`<TEXT variable name> ( <position of the character> )`

If the above program had included the line `PRINT STR(22)`, the character 'x' would have been displayed.

From this example, it is clear that F-Basic allows all text scalars to be treated as a one-dimensional array of characters. Had `STR` been declared as a `TEXT` array, however, the FastCom compiler would have treated the reference `STR(22)` as an array reference and not as a single character substring reference. For further details, refer to the chapter entitled 'Arrays In F-Basic'.

Section 4-More Technical Matters Concerning INTEGERS And REALS

Actually, F-Basic supports 3 different types of integer data specified by the keywords `INTEGER`, `WORD` and `BYTE`. They are declared as shown in the following program segment:

```
INTEGER I,J,k
WORD lm,NP
BYTE some,sum,Sum
```

These three types of integers are represented internally as 32-bit, 16-bit, and 8-bit signed integers respectively. The `WORD` and `BYTE` types are provided to more efficiently utilize data storage space. In general, however, arithmetic involving 32-bit integers is faster than arithmetic with `WORD` and `BYTE` integers.

`REAL` numbers in F-Basic are represented internally using a 32 bit mantissa with no hidden ones and a 16 bit unbiased exponent. This non-standard format has been chosen because it provides increased accuracy while at the same time delivering improved speed of execution when compared with a standard format such as that endorsed by the IEEE. For instance, this format provides 6 digits of significance while the F-Basic format provides 9. The improved speed is evident when comparisons between F-Basic and other Amiga compilers are made using standard real number benchmarks. Examples of several well known tests, including the 'Savage' are available in the timing tests section of the F-Basic sample programs disk provided with your package.

One further advantage of the F-Basic REAL number format is its extended range. For example, the IEEE format referred to above requires that the magnitude of a REAL number be less than 10 raised to the thirty-eighth power (approximately), and valuable physical constants such as the reciprocal of Newton's gravitational constant G are out of range. The F-Basic format, however, will support real numbers in excess of 10 raised to the power ten thousand. This number is larger than the estimated number of elementary particles in the physical universe raised to the power 100. This range even exceeds that provided by the standard IEEE double precision format while retaining approximately 60% of the accuracy. This modest loss of accuracy is compensated for by the significant increase in REAL number arithmetic calculation speed evident with F-Basic.

Section 5-DATA Statements

Declaration statements in F-Basic have two functions. First, they assign a data type to each variable. Second, these statements allocate the memory necessary to store the value of the variable. Normally, the memory allocated for these variables is initialized to zero. It is often useful to have the compiler initialize variables to their correct (non-zero) value. F-Basic provides the DATA statement for this purpose. Its syntax is :

DATA(<variable name>,<literal value or constant>)

Consider the following program segment:

```
PROGRAM DataExample
INTEGER I,J
REAL x
TEXT*30 STRING1,STRING2
DATA(I,22)
DATA(J,10),(x,32.15),(STRING1,"CAT")
DATA(STRING2," ")
```

There are three DATA statements in the above example. As illustrated by the second DATA statement, one may actually have as many variables initialized by a single DATA command as may be accommodated on one line.

There is one very useful and important point to note about using DATA statements on variables declared as type TEXT. If the number of characters in the string assigned as the value of the variable is less than the declared length of the variable, the remaining character

positions are filled with blanks. In other words, the DATA statement 'blank-fills' the remainder of the TEXT variable that would otherwise be uninitialized. For example, STRING1 and STRING2 above have a declared length of 30 characters. The second DATA statement instructs the compiler to place the characters 'CAT' followed by 27 blanks at the memory location assigned to STRING1, while the third DATA line results in the placement of 30 blanks at the memory location assigned to STRING2.

The use of DATA statements to initialize the value of a variable saves both time and space when compared to initializing the variable using an assignment statement. This is because DATA statements generate no executable code.

DATA statements may also be used to initialize arrays and record fields. This use is discussed in the chapters entitled 'Arrays In F-Basic' and 'Records In F-Basic'.

Section 6-CONSTANT Declarations

F-Basic allows variable names to be assigned to a constant value. CONSTANT names differ from ordinary variables only in that they are not allocated any memory space to store their value and, therefore, their value may not be changed. When the FastCom compiler encounters a statement containing a CONSTANT name it merely replaces the name by the assigned value. For instance:

```
PROGRAM ConstantExample
CONSTANT J=3,S=21.4,Family="Smith"
INTEGER I
REAL X
TEXT*10 STR
DATA(I,10)
PRINT I+J,S*S,Family
END
```

Note that the names assigned to CONSTANTS are not declared in any type declaration. The type of a CONSTANT is determined by the value that is assigned to it. The output of the above program would be the integer 13, followed by the real number whose value is 21.4 times 21.4, followed by the characters 'Smith'.

Operators, Expressions, & Assignments In F-Basic

Introduction

This chapter contains a discussion of binary and unary operators on data of type INTEGER, REAL, and TEXT in F-Basic. An explanation of operator precedence and the formation of expressions follows. The chapter concludes with a discussion of assignment statements, the 'backbone' of most computer programs. For quick reference, the appendix of this manual contains a list of all operators supported by the F-Basic language.

Section 1-Binary Arithmetic On INTEGERS And REALS

There are six basic arithmetic operators which are defined for both integer and real values. A list of the F-Basic symbols and corresponding operator names follows:

$\wedge$	Exponentiation
$\backslash$	Remainder
$/$	Divide
$*$	Multiply
$-$	Subtract
$+$	Add

These operators are listed from highest to lowest precedence. This concept is more fully explained in Section 9 of this chapter.

Unlike Pascal and many other modern languages, F-Basic supports mixed-mode arithmetic. This means that one of the two operands may be of type INTEGER, while the other is of type REAL. The resulting type of a mixed mode operation is always REAL.

The operators listed above have the meaning that they possess in ordinary arithmetic. For example:

$$\begin{aligned}2^3 &= 8 \\6.0/2 &= 3.0 \quad (\text{mixed mode}) \\5.14 + 2.17 &= 7.31\end{aligned}$$

The remainder operator is probably less familiar. The value of the

expression $X \setminus Y$ is the remainder upon dividing X by Y. For example:

$$\begin{aligned} 23 \setminus 4 &= 3 \\ 17 \setminus 5 &= 2 \\ 32 \setminus 8 &= 0 \end{aligned}$$

If X and Y are type REAL, the remainder is computed by truncating the quotient X/Y to an integer and then computing the remainder. As an F-Basic formula, the remainder can be expressed as:

$$X = Y * (\text{INT}(X/Y)) + (X \setminus Y)$$

Here, INT is an F-Basic library function that returns the whole number or integer portion of a real number. For instance:

$$3.95 \setminus 1.20 = 0.35$$

$$\text{since } 3.95 = 1.20 * 3 + 0.35$$

There are two special features of computer arithmetic which programmers should bear in mind. First, the result of dividing one integer by another is always an integer. For example, the value of $5/2$ is 2 and NOT 2.50, while $5.0/2$ or $5/2.0$ or $5.0/2.0$ each give 2.50. Again, $1/3$ is 0 and NOT 0.333. Second, a negative real number raised to a real exponent is illegal and will result in an execution error. (If X and Y are type REAL, then X^Y is computed as $\text{EXP}(Y * \text{LN}(X))$. Here EXP is the F-Basic library function for the natural exponent, while LN is the F-Basic library function for the natural logarithm. If X is negative, then Run-Time Error Message #7 concerning an illegal negative argument in the LN function will be invoked, causing execution to cease.)

Many programs involving integer arithmetic do not require the ability to perform calculations that involve large numbers (32 bit integers). The execution of such programs can be speeded up by the use of the &QUICK 1 compiler directive. Any integer arithmetic operations that are performed after this directive is encountered will use faster instruction sequences to implement multiplication and division. As the multiplication and division of integers takes approximately 18 and 40 times as long, respectively, as addition, it is important to speed these operations whenever possible. The &QUICK directive allows the compiler to generate faster code by making two assumptions:

1. In any multiply, both operands have a magnitude less than 32,767. Thus, the result of the multiplication may be as large as 1,073,676,289 and still take advantage of this improved execution code.
2. In any integer division, the denominator and the quotient

both have magnitudes less than 32,767.

For example, 31000×27000 or $1,000,000/21,000$ would be correctly computed after an &QUICK 1 directive has been given, whereas $2 \times 40,000$ and $70,000/2$ would not be correctly computed in these circumstances. One may disable this option at any place in the source code by giving the directive &QUICK 0. In the absence of other &QUICK directives, this value is the default.

The majority of user programs are likely to satisfy the above restrictions, and the use of the &QUICK option can significantly increase performance. F-Basic's extended integer arithmetic and &QUICK options are in contrast with most programming languages for PC's, which implement integer arithmetic by constraining all values to be at most 32,767 in magnitude.

Within the scope of an &QUICK 1 directive, the F-Basic system does not perform run time error checking for division by zero.

Section 2-Relational Operators On INTEGER, REAL And TEXT

A relational operator is one that returns a value of true or false. Relational operators in F-Basic return true as the integer value 1 and false as the integer value 0. There are six relational operators in F-Basic:

<	Less Than
>	Greater Than
=	Equality
<=	Less Than or Equal
>=	Greater Than or Equal
<>	Not Equal

When the operands are of type INTEGER or REAL, the relationals have the usual interpretation. When the operands are of type TEXT, there are two cases to consider. When the strings are of equal length (declared length), the operands are compared lexicographically or in dictionary order. When the strings are of unequal length (declared length), inequality returns the value true and all other relationals return the value false. Thus:

while "CAT"<"Dog" returns the value 1
"CAT"<"Dogs" returns the value 0.

The fact that relationals return integer values allows relational

operators to be embedded in arithmetic expressions. For instance:

$$3 + 2*(X<Y)$$

has the value 5 if X is less than Y, and 3 otherwise.

Section 3-Logical Operators

Logical operators are restricted to logical or integer values. There are three binary logical operators in F-Basic. They are:

XOR	Exclusive Or
AND	Logical And
OR	Logical Or

If A and B have logical values (either 0 or 1), then A XOR B is true when exactly one of A or B is true. The logical expression A AND B is true when both A and B are true. Finally, A OR B is true when either A is true or B is true. These expressions evaluate to false if these conditions are not met.

These three operators may also be used with integers whose value is other than 0 or 1. Suppose that A equals 3 (which is 011 in binary) and B equals 7 (111 in binary). Then A XOR B has value 4 (100 in binary), A AND B has value 3 (011 in binary), and A OR B has value 7 (111 in binary). This use of the operators is said to be a 'masking' operation, meaning that the logical operations are performed on corresponding bits of the operand to yield the corresponding bits of the result.

Finally, the AND and OR operators, when used in a control structure, operate in 'short-circuit' mode. This means that if evaluation of the left operand determines the outcome of the logical operation, then the right operand is not evaluated. This process speeds program execution considerably. For example, the use of the AND operator in the expression (X<0) AND (Y<Z) illustrates the desirability of short circuit operations. The value of the expression must be false if the value of X is greater than or equal to 0. In this circumstance, the expression (Y<Z) will not affect the final value of the entire expression and, therefore, need not be evaluated. Control statements are presented in the chapter entitled 'Control Statements in F-Basic' and include IF statements, WHILE statements, and UNTIL statements as examples.

The FastCom compiler automatically determines by the use of the

expression whether the 'masking' or 'short-circuit' interpretation of these logical operators should be used.

Section 4-The MAX And MIN Operators

MAX and MIN operate on either integer, real, or text-valued expressions. MAX returns the 'larger' of its two operands and MIN returns the 'smaller'. For example:

```
3 MIN -5 = -5
3.1 MIN 2.7 = 2.7
"CAT" MAX "FAT" = "FAT"
```

Section 5-Shift Operations On INTEGERS

F-Basic supports 4 shifts which operate on integers. The syntax is :

<value to be shifted> <shift operator> <number of bits to be shifted>

The shift operators are:

ASL	Arithmetic Shift Left
ASR	Arithmetic Shift Right
LSL	Logical Shift Left
LSR	Logical Shift Right

ASR and LSR, while both shifting the left operand right, differ in that LSR fills the bits at the left end of the integer with zeros while ASR fills with zeros if the value being shifted is positive and fills with ones if the value being shifted is negative. ASL and LSL do not differ, but are both included for the sake of symmetry. These operators mimic the actions of the 68000 assembly language mnemonics of the same name.

Shifting positive integers to the left or right is an efficient way to perform multiplication or division by a power of 2. For instance, 111 LSL 3 is equivalent to but faster than, 111*8. 111 LSR 4 is equivalent to but faster than, 111/16.

Section 6-String Concatenation

The concatenation operation, denoted by &, may be used to combine two strings into a longer string. Thus, "FAT"&"CAT" produces the string "FATCAT".

Section 7-Unary Arithmetic And Unary Logical Operators

F-Basic provides two unary arithmetic operators, + and -, and one unary logical operator, NOT. All unary operators have a higher precedence than any binary operators. If A represents a real number or integer, then the expression +A has the same value as A, while -A is the negative of A. The logical operator NOT, when applied to a logical value (0 or 1) produces logical negation. NOT of a TRUE expression is FALSE, and NOT of a FALSE expression is TRUE. NOT, when applied to an expression whose value is other than 0 or 1, changes the value to 0 or FALSE.

Section 8-PEEK, POKE, And Address Operators

The @ operator in F-Basic, when applied to any variable name, array name, or array reference returns the address of that variable. The syntax is:

```
@ <variable/array name>  
@ <array name>(<subscript>)
```

Array names and array references are discussed in the chapter entitled 'Arrays In F-Basic'.

For example, if SUM represents an INTEGER variable whose value is stored beginning at address 100000, then @SUM would evaluate to 100000. The @ operator serves the same purpose as the VARPTR function in most BASIC packages.

Many BASIC systems allow the storage and retrieval of data from specified memory locations using the POKE and PEEK commands. In such languages, the following syntax is typical:


```
POKE <address>,<value>
PEEK <address>,<value>
```

F-Basic supports the same features but implements them in a more flexible and powerful way. Three unary operators are supported: #, \$, and ^ to designate long-word (32 bit), word (16 bit), and byte (8 bit) respectively.

First, consider the F-Basic equivalent of POKE. There are three possible forms:

```
#<even address> = <integer value>
$<even address> = <integer value>
^<address> = <integer value>
```

Here, <address> represents any integer-valued expression which is interpreted as the address where <integer value> is to be stored. The <integer value> will then be stored respectively as long-word, word, or byte. For long and word operations an <even integer> is required or a system crash will occur. Thus, when #, \$, or ^ appear as the first character in an assignment statement, they implement respectively POKEL, POKEW, and POKE.

An extended form of the POKE operation that is supported by F-Basic is:

```
^ <address> = <text valued expression>
```

This stores the entire text string on the right in memory beginning with the address given by <address>.

Another extended form of the POKE operation that is supported by F-Basic is:

```
# <even address> = <real valued expression>
```

This stores the real number at the memory location specified by <even address>. As real numbers in F-Basic occupy two long words, 64 bits are actually transferred to memory with this operation.

One CAUTION is in order. When modifying memory locations using the operators explained above, one should normally only modify memory that is part of the data segment of the current program or memory that has been allocated with either the ALLOCATE or ALLOCATE_WITH library functions available in F-Basic. Otherwise, parts of the Amiga-DOS operating system, the ROM Kernel, or the screen display memory may be destructively modified producing unpredictable results.

When #, \$, or ^ occur as unary operators in any other position in the line, they implement PEEKL, PEEKW, and PEEK respectively. This means that a long-word, word, or byte is retrieved from memory and used as part of the expression in which the operators occur. The syntax applicable is:

```
# <even address>
$ <even address>
^ <address>
```

Here, again, <address> is an integer-valued expression which specifies the memory location from which the data is to be retrieved. As before, an <even integer> is required when using # and \$ or a system crash will result. Two examples are presented to illustrate these operations. The first example prints out memory by byte from address 100000 to address 100020:

```
PROGRAM PrintMem
INTEGER I
? Comment: Fetch and print byte from location I
FOR I=100000 TO 100020
    PRINT ^I
NEXT I
END
```

The next example illustrates the power of embedding the PEEK operation within expressions:

```
PROGRAM PeekExamp
REAL X,Y
INTEGER I,J,K,L
I=@J
J=@K
K=@L
L=10
PRINT #(#(#I)) + 10
END
```

This program prints the integer 20. This can be seen by observing that the inner-most # returns the contents of the variable J, as I contained the address of J. The middle # sign thus returns the contents of K, as J contained the address of K. The contents of K is the address of L, thus the outer # returns the contents of L or 10. The final answer of 20 comes from this 10 and the additional 10 added to the expression.

The term 'precedence' refers to the order in which operations are performed in the absence of parentheses. In simple expressions such as $3*5+7$, the reader would naturally perform the multiplication before the addition. This is a reflection of the higher precedence given to multiplication when compared with that given to addition. If it is desired that the addition be performed before the multiplication, then parentheses must be used in the expression to override the rules of precedence, as in $3*(5+7)$.

The rules of precedence may be summarized as follows:

1. In the absence of parentheses, operators of higher precedence are evaluated before operators of lower precedence.
2. In the absence of parentheses, if two operations have the same precedence, then the leftmost operator of the two is evaluated first.
3. The 'level' of an operator is an integer that specifies the number of balanced left/right parentheses pairs which contain the operator. Operators of a higher level are evaluated before operators of a lower level.

Consider the expression $5-6-7$. Both subtraction operations are of equal precedence. F-Basic thus evaluates the leftmost operator first, obtaining -8 for the answer. If the rightmost minus had been evaluated first, then the reported value would have been +6.

To illustrate the last rule of precedence, consider the expression $7/(6*(7-8)+9)$. Here, the division operator occurs at level 0. The multiplication and addition operators are at level 1, and the minus operator is at level 2. The evaluation thus is performed in the order -, *, +, and / (integer division), obtaining the value 2. The same expression without any parentheses would be evaluated in the order /, *, -, and +, obtaining the value 8.

Throughout this manual, the term expression has been used. A precise definition of an expression is presented for the sake of completeness.

1. Any variable, or any literal is an expression.
2. Enclosing any expression in parentheses results in an expression.
3. A unary operator acting on an expression results in an expression.
4. Combining any two expressions with a binary operator produces an expression.

A complete precedence table for all valid binary and unary operators in F-Basic can be found in the appendix entitled 'Precedence Table For F-Basic Operators'.

Section 10-Assignment Statements And The Immediate Assignment Operator

The most basic command in most programming languages is the assignment statement. This causes a value to be stored at a memory location specified by a variable. The syntax is:

`<variable name> = <expression>`

If `<variable name>` represents a variable declared to be of type TEXT, then `<expression>` must also be string valued. If `<variable name>` represents a variable declared to be of type INTEGER or REAL, then `<expression>` may be of either type INTEGER or REAL. Thus, F-Basic permits REAL variables to be equated to integer expressions and INTEGER variables to be equated to real expressions. If the variable is of type INTEGER, and the expression is of type REAL, the value of the expression is truncated to be an integer. If the variable is of type REAL, and the expression is of type INTEGER, the value of the expression is converted to a real number with the same magnitude. Thus:

```
PROGRAM MixedMode
INTEGER I
REAL X
I=2.3
X=16
PRINT I,X
END
```

will store the integer 2 in variable I, and the real value 16.0 in variable X.

F-Basic also supports multiple equates. In a multiple equate, all of the variables must be of the same type. For example, the program:

```
PROGRAM Multiple
INTEGER I,J,K,L,M,N,o,p
I=J=K=L=M=N=o=p=2
PRINT I,J,K,L,M,N,o,p
END
```

will print eight 2's on the display screen.

F-Basic allows a portion of an expression to be assigned to a variable of type INTEGER. The operator is ! and the syntax is:

<integer expression> ! <INTEGER variable name>

For example:

```
PROGRAM ImmedAssign
INTEGER I,J
I=5
J=3*(I+4)!I + 2
PRINT I,J
END
```

will display 9 for the variable I and 29 for the variable J. By referring to the operator precedence table provided in the Appendix, it is evident that the immediate assignment operator ! has a precedence higher than all other binary operators. In other words, in the absence of overriding parentheses, ! is performed first. The immediate assignment operator is particularly useful when used with F-Basic control structures, which is the subject of the next chapter.

The immediate assignment operator should not be used within a PRINT statement.

Control Structures In F-Basic

Introduction

As discussed in the chapter entitled 'F-Basic Program Structure', an F-Basic program is divided into two sections, the declaration part and the executable part. Execution of the program begins with the first line of the executable part and continues line by line, in the absence of control structures, through the program. Thus, the purpose of a control structure in an F-Basic program is to alter this execution sequence.

Section 1-Statement Labels, Statement Numbers, And The GOTO Statement

Any executable statement in F-Basic may be prefixed by a statement label or statement number. This label or number is used by other program lines to refer to the statement to which it is attached. A statement label is any sequence of characters that begins with an underline symbol or letter of the alphabet and contains only letters, digits, or underline characters. This is the same requirement that a valid variable name in F-Basic must satisfy. The syntax to label a program line with a statement label is:

```
{ <statement label> } <executable line>
```

The use of the brace pair '{}' allows the FastCom compiler to distinguish between the desired statement label and other program statements or variable names. For example:

```
{Compute_N} N=5*a+b
```

attaches the label 'Compute_N' to the program statement $N=5*a+b$. Another acceptable form is:

```
{Compute_N} CONTINUE  
N=5*a+b
```

The CONTINUE statement has no effect and serves only as a marker for a label. A final acceptable form is:

```
{Compute_N}
```


N=5*a+b

By placing the statement label by itself above the desired line, the same labeling effect is achieved.

A statement number is any sequence of digits; the same rule used to specify an integer literal. As statement numbers are in no danger of being confused with variable names, they are not enclosed in a brace pair. The program line illustrated in the last example could instead be labeled with statement number 100 using the following:

100 N=5*a+b

or, equivalently:

100
N=5*a+b

It is a good practice to use statement labels instead of statement numbers, as their names may be chosen to convey information about the purpose of the lines that they label. Certainly, a label of either kind is only necessary on a program line if control is to be passed to the line from some other point in the program. Keeping the number of labeled statements to a minimum in this way makes those labels that are used easier to locate when examining a program.

The simplest control structure in F-Basic is the GOTO statement. The syntax is:

GOTO <statement label or statement number>

For example, either of these lines:

GOTO Compute_N
GOTO 100

will cause the statement N=5*a+b, as defined above, to be the next statement executed.

There are two restrictions on the use of the GOTO statement. One is simple, and one is more technical. First, the GOTO statement and the statement to which it passes control must be within the same program unit. In other words, they must be in the same main program or same subprogram. Secondly, these two statements must be within 32,767 bytes of one another in the executable code generated by the compiler. A typical F-Basic program statement might occupy 20-25 bytes and thus, this restriction means that a GOTO statement must be within approximately 1500 lines of its target. The FastCom compiler will display an error message during compilation if either of these two conditions is not met.

Section 2-GOSUBS And Local Subprograms

F-Basic, like most other BASIC packages, supports 'local' subprograms. A local subprogram is a sequence of program lines beginning with a labeled statement, and terminating with the RETURN or LRETURN statement. The term local implies that the subprogram is part of the same program unit as the statement that transfers control to it. External subprograms of type SUBROUTINE and FUNCTION, typically provided in more advanced high level languages, are supported by F-Basic and discussed in the chapter entitled 'Subprograms In F-Basic'.

The program line which transfers control to a local subprogram is the GOSUB statement. The syntax is:

GOSUB <statement label or statement number>

The following program illustrates the use of the GOSUB and LRETURN statements. It will input the initial value of a certificate of deposit, the annual interest rate, and the number of years before maturity. It uses a local subprogram to compute the final value of the CD, assuming annual compounding of interest.

```
PROGRAM ItAddsUpFast
INTEGER NYears
REAL InitialValue,YRate,FinalValue
PRINT "Enter Years, Initial Value, Interest Rate (as in 8.5)",
INPUT NYears,InitialValue,YRate
GOSUB Compute_Final_Value
PRINT "The Final Value of The CD Is:",FinalValue
STOP
{Compute_Final_Value}
  FinalValue=InitialValue*(1.0+YRate/100.0)^NYears
  LRETURN
END
```

Clearly, a local subprogram in this example is unnecessary, but it serves to illustrate the use of the GOSUB and LRETURN or RETURN lines. When the GOSUB is encountered, control is passed to the statement labeled 'Compute_Final_Value' and execution continues there until the LRETURN statement is encountered. At this point, control returns back to the first statement following the GOSUB, which is the PRINT line in this example. Care must be taken to insure that a program always enters a local subprogram through the use of a GOSUB

(or ON ... GOSUB) statement, as a program which inadvertently 'falls' into a local subprogram will LRETURN upon completion of the subprogram to an unpredictable and undesirable location. This is avoided in the sample program by using the F-Basic STOP command. STOP is similar to END in that when either are encountered in the main program, execution terminates. STOP is different than END in that it may be used anywhere in the main program as often as desired while END can be used only as the last statement of a program unit. STOP can only be used in the main program.

Section 3-IF Statements

The simplest form of the F-Basic IF structure is the 'one-line IF'. The syntax is:

```
IF <condition> THEN <executable line>
```

Keywords in F-Basic, such as IF, THEN, GOTO, GOSUB, etc. must be delimited by blanks, that is there must be at least one blank space separating the keywords from other characters on the line. In the above syntax, <condition> represents an expression returning a logical or TRUE/FALSE condition. If the condition is TRUE, the <executable line> is executed, otherwise it is skipped. In either case, control then passes to the first statement following the one-line IF. For example:

```
IF N<10 THEN I=N+4-Sum_Of_Digits
```

The next simplest F-Basic IF structure is the 'block IF'. The one-line IF conditionally executes a single statement, while the block IF conditionally executes a group of statements. The syntax is:

```
IF <condition> THEN
    <any number of executable lines>
ENDIF
```

When <condition> is TRUE, the block of statements between the IF and the ENDIF statements are executed sequentially. If <condition> is FALSE, this block is skipped. In either case, control then passes to the first program line following the ENDIF statement. For example:

```
IF N<10 THEN
    I=N+4-Sum_Of_Digits
    J=4*N+32*Today's_Date
    K=1
```


ENDIF

When a programmer wishes to execute one of two alternative sequences of statements, the IF/THEN/ELSE/ENDIF structure should be used. The syntax is:

```
IF <condition> THEN
    <first sequence of executable lines>
ELSE
    <second sequence of executable lines>
ENDIF
```

For instance:

```
IF N<10 THEN
    I=N+4-Sum Of Digits
    J=4*N+32*Today's Date
    K=1
ELSE
    I=N+1
    J=N+32
ENDIF
```

produces a structure such that when <condition> is TRUE, the first sequence of lines is executed and the second is skipped. When <condition> is FALSE, the first sequence is skipped and the second is executed. In both cases, control then is passed to the first statement after the ENDIF statement.

When there are more than two alternate sequences of statements to be executed, one may choose the IF/THEN/ELSEIF/ELSE/ENDIF structure. The syntax is:

```
IF <condition1> THEN
    <sequence 1 of executable lines>
ELSEIF <condition2> THEN
    <sequence 2 of executable lines>
ELSEIF <condition3> THEN
    <sequence 3 of executable lines>
    .
    .
ELSE
    <final sequence of executable lines>
ENDIF
```

Actually, in the above syntax, the presence of the ELSE with its accompanying sequence of lines is optional. There may be any number of ELSEIF <condition N> blocks in the structure. In the format shown, if <condition1> is TRUE, then sequence 1 is executed, and the remaining

sequences are skipped. If <condition1> is FALSE, sequence 1 is skipped, and <condition2> is evaluated. If this is TRUE, sequence 2 is executed, and the remaining sequences are skipped. If <condition2> is FALSE, sequence 2 is skipped, and <condition3> is evaluated. This process continues until one of the ELSEIF conditions succeeds or until the ELSE is encountered. In the case that the ELSE is encountered, the last sequence is executed. If no ELSE is present, and all of the conditions are FALSE, then none of the sequences are executed.

For example, suppose a tax schedule assigns three tax brackets to taxpayers. For gross family incomes less than \$30000., a tax rate of 15% applies. For incomes between \$30000. and \$40000., a marginal rate of 20% applies, and for those earning over \$40000., a marginal rate of 25% is assessed. The following IF/THEN/ELSEIF/ELSE/ENDIF structure could be used to compute the tax owed by the taxpayer:

```
IF INCOME<30000.0 THEN
    Tax=0.15*INCOME
ELSEIF INCOME<40000.0 THEN
    Tax=0.20*(INCOME-30000.0)+4500.0
ELSE
    Tax=0.25*(INCOME-40000.0)+6500.0
ENDIF
```

All IF structures in F-Basic may be nested. This means that any of the sequences of statements referred to above in an IF, ELSE, or ELSEIF block may themselves contain other IF structures. Each of these nested IFs may themselves contain nesting of IF structures. In this way, any set of alternative conditions can be specifically programmed.

The reader is also reminded that the <conditions> described above and in the control statements below may in fact be multiple conditions separated by the F-Basic operators AND, OR, XOR, and NOT. As was discussed in the chapter entitled 'Operators, Expressions, & Assignments In F-Basic', these multiple conditions are evaluated using 'short-circuit' methods, thereby delivering the fastest possible execution times. This is particularly important in light of the fact that a tremendous amount of time in most programs is spent making decisions about the proper sequence of statements to execute given the current condition of program variables.

Section 4-The WHEN Statement (Case Structures)

An alternate method of selecting one of several possible sequences of executable lines is the WHEN structure. The syntax is:


```

WHEN <selector expression> IS
    [ <value1> ]    <sequence 1 of executable lines>
    [ <value2> ]    <sequence 2 of executable lines>
    .
    OTHERWISE
        <final sequence of executable lines>
ENDCASES

```

The <selector expression> is any expression of type INTEGER or TEXT. Each of the <value n> quantities must be of the same type as the <selector expression>. When the WHEN statement is encountered, the <selector expression> is evaluated. If the result is equal to any of the <value n> expressions, the sequence of lines corresponding to that value is executed and all other alternative sequences are omitted. If the value of the <selector expression> does not equal any of the possible <value n> alternatives, then control passes immediately to the final sequence of lines in the OTHERWISE block, if present. The OTHERWISE block is optional, and if it is not present and no match is found, all sequences corresponding to the various <value n> blocks are skipped.

Actually, the '[' pair may contain a list of target values separated by commas. For example:

```

[1,2,3,7,8]    <sequence of executable lines>

```

will be executed if the <selector expression> for this WHEN or Case structure has the value 1, 2, 3, 7 or 8. A complete example of a WHEN block is:

```

WHEN 3*I+4 IS
    [0,1,2]    J=14
                K=I+13
    [Tmap^2]    J=I+14
                K=K*K+J*J
    OTHERWISE
                Tmap=14
                J=K=0
ENDCASES

```

Any WHEN structure can be represented by an equivalent IF/THEN/ELSEIF/ELSE/ENDIF structure. WHEN blocks are somewhat less general than their IF block counterpart, as the selection of which alternative sequence of statements to execute is based on equality tests, whereas the extended IF structure will perform selection based upon any type of condition. Some programmers, however, prefer the WHEN statement when applicable because it is thought to produce more

readable code.

As a matter of taste, the final statement in a WHEN block may be either ENDCASE or ENDCASES. Both have the same desired effect.

Section 5-FOR Loops And The EXIT Statement

In the previous control structures, one sequence of executable lines chosen from several possible sequences was executed once. The FOR loop control structure allows the repeated execution of a single sequence of executable lines. The simplest form of the FOR statement is:

```
FOR <variable name> = <initial value> TO <final value>
    <sequence of executable lines called the 'Loop Body'>
NEXT <variable name>
```

Here, <variable name> is called the loop 'index' or loop 'counter'. It is initialized to <initial value> and a first pass through the Loop Body is made, assuming that the <initial value> is less than or equal to the <final value>. When the NEXT statement is encountered, the value of the loop index is compared with <final value>. If the value of the loop index is greater than or equal to <final value>, execution of the Loop Body is discontinued, and control passes to the first statement after the NEXT line. Otherwise, the value of the loop index is increased by one, and the Loop Body is again executed. If <initial value> is originally greater than <final value>, the Loop Body is skipped. The following example computes the value $Sum=1+2+3+4+...+N$, for any value of N specified:

```
PROGRAM Summation
INTEGER Sum,I,N
Sum=0
INPUT "Enter N" N
FOR I=1 TO N
    Sum=Sum+I
NEXT I
PRINT "The Sum Is",Sum
END
```

In this program, the Loop Body consists of only a single statement. In general though, the Loop Body may contain any number of lines, including other control statements such as IFs and other FORs. 'Nesting' of FOR loops is an extremely useful programming tool. F-Basic allows a single NEXT to terminate several FOR loops when they

are nested. This is accomplished by appending a list of the loop counter variables, separated by commas, to the NEXT statement. For instance, the line NEXT I,J will terminate the two most recently defined FOR statements.

In the above syntax, <initial value> and <final value> may be expressions.

One may never directly transfer control from a statement outside of the FOR loop to a statement inside the Loop Body. The FastCom compiler creates the FOR loop in a way that requires execution to flow normally through the actual FOR statement at the top of the loop. Thus, any attempt to jump into the middle of the loop will have undesirable and unpredictable results.

One may, however, directly transfer control from the inside of the FOR loop to a program line outside of the loop using a GOTO statement. The FastCom compiler will allow this usage of the GOTO provided that the loop in which it is placed is not contained within other FOR loops. In other words, the FOR loop containing the GOTO may not be nested within other FOR loops.

Additionally, F-Basic supports the EXIT statement within the Loop Body of any FOR loop (nested or not). This statement, when executed, causes an immediate and unconditional transfer of control to the first program line after the NEXT statement of the loop containing the EXIT. The EXIT statement must always be used to prematurely leave the Loop Body of a nested FOR loop.

An alternate form of the FOR loop permits the loop index to be incremented by an arbitrary value (other than 1 as in the standard FOR loop above) before repeated execution of the Loop Body proceeds. The syntax is :

```
FOR <variable name> = <initial value> TO <final value> STEP <step size>
                        <Loop Body>
NEXT <variable name>
```

The number of times that this loop is repeated depends upon the value of an expression called 'Control':

$$\text{Control} = ((\text{<final value>} - \text{<initial value>})/\text{<step size>})$$

If Control is negative, the loop is skipped. If the value of Control is zero, and <step size> differs in sign from (<final value>-<initial value>), then the loop is also skipped. Otherwise, the Loop Body is executed Control+1 times. The value of <variable name>, initially set to be <initial value>, is changed after each pass through the Loop Body by an amount equal to <step size>, except on the final pass

through. For example:

```
FOR I=30 TO 10 STEP -6
    <Loop Body>
NEXT I
```

will be executed 4 times. The value of I on each pass will be 30, 24, 18, and 12 respectively, and will remain 12 upon leaving the loop.

F-Basic also supports the use of a loop index declared to be of type REAL. This is particularly useful in some scientific programs. FOR loops of this type follow the same execution rules described for integer valued FOR loop indices.

If the loop index is declared as an INTEGER variable, then <initial value>, <final value>, and <step size> must be integer valued expressions. If the loop index is declared as a REAL variable, these quantities may be either real valued or integer valued, and are evaluated according to the mixed mode rules discussed in Chapter 4. Because a loop with an index of type INTEGER is considerably faster than the corresponding loop with an index of type REAL, real valued loops should be used only when there is a compelling reason to do so.

As a programming convenience, F-Basic permits the use of a loop index whose type has not been previously declared provided that the Loop Body contains no other FOR loops. In this case, the loop index defaults to type INTEGER, and is only defined within this Loop Body.

Section 6-The WHILE Statement

A second looping structure in F-Basic is the WHILE statement. Its syntax is:

```
WHILE <condition> DO
    <sequence of executable lines called the 'While Body'>
ENDWHILE
```

Any time that the WHILE statement is encountered, <condition> is evaluated. If the result is TRUE, the While Body is executed. If the result is FALSE, control passes immediately to the first program statement after the ENDWHILE statement, and the While Body is skipped. When the ENDWHILE statement is encountered, control is unconditionally transferred back to the WHILE statement. For example, to calculate the same sum presented in the previous section using the WHILE statement, one could write:


```

I=1 ; Sum=0
WHILE I<=N DO
    Sum=Sum+I
    I=I+1
ENDWHILE

```

The presence of the keyword DO in the WHILE statement is optional. Branching into or out of WHILE loops is permitted without restriction.

Section 7-The REPEAT Statement

The final looping structure in F-Basic is the REPEAT structure. Its syntax is:

```

REPEAT
    <sequence of executable lines called the 'Repeat Body'>
UNTIL <condition>

```

When the REPEAT statement is encountered no action is taken. REPEAT merely marks the beginning of the Repeat Body. At this point, the statements of the Repeat Body are executed sequentially. When UNTIL is encountered, the <condition> is evaluated and if the result is FALSE, control passes to the REPEAT statement. If the result is TRUE, control passes to the first statement after the UNTIL.

The REPEAT structure differs from the WHILE structure in two major respects. First, the WHILE statement tests <condition> before entering the While Body, whereas the REPEAT statement tests <condition> after executing the Repeat Body. Secondly, if the WHILE's <condition> is TRUE, the While Body is executed, whereas if the UNTIL's <condition> is TRUE, control passes away from the REPEAT structure to the first statement after the loop.

One could compute the sum referred to in the last two sections with a REPEAT statement as follows:

```

Sum=0 ; I=1
REPEAT
    Sum=Sum+I
    I=I+1
UNTIL I>1

```

One may transfer control into or out of a REPEAT block without

restriction.

Section 8-The ON...GOTO And ON...GOSUB Statements

A final method for selecting between alternate blocks of statements involves the ON structures. The ON GOTO statement has syntax:

ON <integer valued expression> GOTO <list of statement labels or statement numbers separated by commas>

For example:

```
ON I+K GOTO 100,200,TryAgain,400
```

satisfies the syntax. When the ON GOTO statement is encountered, the <integer valued expression> is computed. If the result is zero, negative, or greater than the number of statement labels or statement numbers in the list (4 in the above example), then the ON GOTO statement has no effect, and control passes to the next statement. Otherwise, control passes immediately to the label or statement number whose position in the list corresponds to the value of the expression. In the example, if I+K is 1, control will pass to the program line bearing statement number 100. If I+K is 2, control will pass to the statement bearing statement number 200. If I+K is 3, control passes to the statement labeled TryAgain, and if 4, to the line with statement number 400.

The syntax and behavior of the ON GOSUB statement is similar to the ON GOTO statement:

ON <integer valued expression> GOSUB <list of statement labels or statement numbers separated by commas>

The only difference is that the ON GOSUB causes control to be passed to a local subprogram if the value of <integer valued expression> is in the range defined by the number of labels or statement numbers in the list. Thus, the statement numbers or statement labels must prefix the initial line of a local subprogram. Control will eventually return to the first statement after the ON GOSUB line when the local subprogram's RETURN or LRETURN statement is executed. Further details about local subprograms are contained in Section 2 of this chapter.

The use of the ON structures is more difficult than the corresponding extended IF or WHEN structures. However, as the number

of alternatives increases, extended IF or WHEN blocks take progressively longer to locate the correct alternative. The time required to execute an ON statement is independent of the number of choices possible. The break even point between the use of an ON structure and the use of an extended IF or WHEN block is typically 3 alternatives.

Section 9-Conclusion

All multiple statement control structures may be nested in F-Basic. The only one with restrictions on the transfer of control into or out of the structure is the FOR loop. Of the three looping structures supported by F-Basic, the FOR loop is the most efficient in terms of overhead.

Each of these control structures confine their effects to the program unit in which they occur. Therefore, they may not be used to transfer control between distinct program units. All labels or statement numbers referenced by the structure must occur within the current program unit.

F-Basic provides this wide selection of control structures to accommodate differing programming styles and tastes.

Keyboard Input And Screen Output In F-Basic

Section 1 - Keyboard Input

Keyboard input may be obtained by using the INPUT statement. The correct syntax follows:

INPUT <list of variables separated by commas>

When this statement is executed, a '?' prompt is displayed on the screen so that the user is alerted that input is expected. Optionally, the programmer may choose to supply an additional text statement to prompt the user about the specific type of input information desired. The correct syntax for the optional text prompt is:

INPUT <literal prompt string> <list of variables separated by commas>

Each input list represents data that the user will enter on a single line. The following program segment illustrates both forms of the input statement:

```
PROGRAM SampleInput
INTEGER A,NUM
REAL X,Y
TEXT*20 STRING1,STRING2
INPUT A
INPUT "ENTER NUM" NUM
INPUT "Enter a Real Number and Two Text" X,STRING1,STRING2
```

The first INPUT statement has no prompt string, and the input list contains a single integer variable A. When this statement is executed, only the '?' prompt will appear on the left of the screen and the user should enter a single integer followed by a carriage return. When the second INPUT statement is executed, the system will print the contents of the prompt string, 'ENTER NUM', followed by the '?' at the current location of the cursor. The user again should enter a single integer followed by a carriage return. The third INPUT statement contains a prompt string and a list containing three variables. When executing this statement, the system will print the prompt statement followed by the '?', and then expect three separate data items to be entered on a single line, terminated with a carriage return. Since more than one item is expected, the system must have a way of telling where each item begins and ends. This is done through the use of the separator character, which is a special character that

is not part of the data item being entered. The default value of the separator is the comma character ','. Thus, in response to the third INPUT statement one might enter:

Enter a Real Number And Two Text? 2.15,CAT,Dog

When this input is processed by the system, the value of X will become 2.15, the value of STRING1 will become "CAT", while that of STRING2 will become "Dog".

There are three important points to be made with regard to character string input such as was just displayed for the variables STRING1 and STRING2. First, when text literal character strings occur within an F-Basic program, they must be enclosed or delimited by the quotation mark " (or the current value of the compiler variable DELIMITER, as explained earlier). However, when character strings are entered from the keyboard via the INPUT statement they are not enclosed with quotes as with CAT and Dog above. Second, all text variables in the INPUT statement are completely filled with blanks by the system before the user's response is used. Finally, INPUT of text variables may not use all of the characters entered by the user if the declared length of the text variable is not large enough to store them all. For example, STRING1 and STRING2 were declared to be 20 characters wide. If the user had typed 'GollyGollyGollyGollyGolly' instead of CAT, only the first 20 characters 'GollyGollyGollyGolly' would have been stored in STRING1.

The comma is adequate as a separator in most situations. However if the character string entered by the user contains an embedded comma the system will incorrectly terminate input for that variable when the comma is encountered. For example, one might desire that STRING1 be filled through the INPUT statement with the fourteen character string 'We, The People'. If the value of the separator character was the comma, STRING1 would receive only the two character string 'We'. This difficulty may be overcome by changing the value of the separator character. To do this in F-Basic, one need only issue a compiler directive in the program before the appropriate INPUT statement. The syntax is:

```
&SEPARATOR <single character>  
or &SEPARATOR #<base 10 integer>
```

The second form of the directive allows the user to specify the base 10 integer equivalent of the ASCII value of the separator character. This is particularly useful when an "unprintable" character is selected to be the separator. The following three statements change the separator character to '{', perform the input, and restore the value of the separator to the comma:


```

&SEPARATOR {
? Comment - Or Equivalently &SEPARATOR #123
INPUT "Enter a Real Number and Two Text" X,STRING1,STRING2
&SEPARATOR ,

```

Although F-Basic supports a wide variety of data types, only six are permitted in the INPUT statement. The compiler will only allow a variable to be used in INPUT if it is one of:

1. Type INTEGER
2. Type WORD (16-bit integer)
3. Type BYTE (8-bit integer)
4. Type PTR TO
5. Type REAL
6. Type TEXT

Section 2 - Unformatted Screen Output

The PRINT statement allows the value of any expression (not just variable names) to be written to the screen. The syntax of PRINT is:

```
PRINT <expression1>,<expression2>,<expression3>,...
```

The statement also allows an optional format statement, with syntax:

```
PRINT <expression1>[format1],<expression2>[format2],...
```

Format specifications are fully discussed in Section 3 of the chapter.

The type of an expression to be printed may be any of the six mentioned at the end of Section 1. Additionally, relational expressions, such as $X < > Y$ may be printed to the screen. The value printed will be one if the expression is TRUE and zero if the expression is FALSE. The following program segment illustrates the use of an 'unformatted' F-Basic PRINT statement:

```

PROGRAM SamplePrint
INTEGER I,J
REAL X
TEXT*6 STR
I=2
J=3
X=2.15
STR="Jockey"

```



```
PRINT I+J,X,STR
```

Before displaying the system's response upon execution of this segment, the term 'field width' must be defined. Field width refers to the number of character positions or screen columns used to display a particular value. In an unformatted print, the default field width of an INTEGER is twelve characters. Expressions of type WORD, BYTE, PTR_TO and Relational also have a default field width of twelve characters. Expressions of type REAL default to 20 characters, while those of type TEXT default to their declared length. Thus, in the above PRINT, the value 5 will appear in the first 12 columns of the display (left filled with blanks), the value of X will appear in the next 20 columns (again left filled with blanks), and the character string 'Jockey' will appear in the next 6 columns. The resulting output on the screen will therefore appear as follows:

```
5          2.150000E+000Jockey
```

Notice that the real number X is displayed in the 'scientific form' or 'exponential format' convention discussed in Chapter 3.

Warning: Suppose that STR had instead been declared as:

```
TEXT STR*10
```

and that the F-Basic statement STR="Jockey" was again present in the program segment. The last four characters of STR are not blank-filled by this assignment and, therefore, their value is unpredictable. The PRINT statement containing STR as an expression would assign a field width of 10 columns to display STR, and thus the last four characters printed could not be predicted in advance. This situation can be avoided in several ways. One is to blank fill the text variable STR with a DATA statement (again, see Chapter 3). Alternatively, one may use a format statement as explained below and direct the system to use a field width of only six characters. Finally, one may direct the system to print only the first six characters of STR using a substring reference, as in:

```
PRINT I+J,X,STR(1:6)
```

A general statement concerning the printing of REAL data should be made before leaving this section. The format in which F-Basic outputs a REAL number provides a maximum of three (3) digits for its exponent. An attempt to print a number as large or larger than 1.0E+1000 will produce unexpected results.

The programmer wishing to alter the default field widths assigned by the system may do so with the format specification eluded to above. In the case of REAL expressions, the programmer may additionally desire that ordinary decimal notation be used instead of scientific notation and/or that more decimal places be displayed. Again, the format specification supported by F-Basic allows this flexibility.

A. Formatting Integer Data

The syntax of an integer format is:

[<integer representing the field width>]

B. Formatting Text Data

The syntax of a format specification for text data is the same as for integers as described in A.

C. Formatting Real Data

The syntax of a format specification for real data is:

[<E or F> <field width> . <# of digits to right of decimal point>]

Recall the example program segment of Section 2. The following modification of the PRINT line will reserve three columns of the line for the integer 5. Additionally, the REAL variable X will be printed in ordinary decimal notation with a field width of five and displaying two digits to the right of the decimal point, and the value of STR will be assigned a field width of ten digits.

```
PROGRAM SamplePrint
INTEGER I,J
REAL X
TEXT*6 STR
I=2
J=3
X=2.15
STR="Jockey"
PRINT I+J[3],X[F5.2],STR[10]
```

The resulting output would appear as follows:

```
5 2.15    Jockey
```


In a real format specification, the <E or F> is optional, and the default is E. The E option denotes exponential format while the F stands for 'fixed point' or ordinary decimal notation. For E formats, the field width specified must be at least 8 larger than the number of digits to the right of the decimal point. This allows adequate columns to properly display the number. For F formats, the field width specified must be at least 3 greater than the number of digits to the right of the decimal point. For F formats, the field width may not exceed 18 and the number of digits to the right of the decimal point should not exceed 10. The FastCom compiler will accept more than 10 in a source program, but will internally reduce this number to 10.

The term 'field width overflow' refers to the situation in which the number of characters required to print the value of an expression is greater than the field width specified by the programmer in the PRINT statement. When a field width overflow occurs, F-Basic alerts the user to this condition by filling the screen columns reserved for that variable with the star character '*'. Thus, if a field width of three is specified in a PRINT statement for an integer variable whose value is 10000, the user would see only three star characters in the space reserved for the integer. Field width overflow checking is supported by F-Basic for all of the seven possible output data types listed above.

Many BASIC languages available for the Amiga achieve format control using the PRINT USING statement. The authors of F-Basic believe that the format specifications outlined above are more compact and easier to use than the PRINT USING statement.

Although the examples in Sections 2 and 3 show the PRINT list as either totally formatted or totally unformatted, one can actually mix formatted and unformatted expressions in a PRINT list.

Section 4 - Line And Blank Control Using PRINT

Normally, the output of one complete print statement terminates with a carriage return being issued by the system, causing the cursor to advance to the next line. The programmer desiring that the output from two or more PRINT statements occupy the same display line is accommodated by F-Basic. The addition of a comma as the last character in the print list instructs the system not to issue the carriage return, and thus the cursor position after printing is undisturbed. Compare the output from the following two simple examples:


```

{Example 1}    FOR I=1 TO 5 ; PRINT I[2] ; NEXT I
{Example 2}    FOR I=1 TO 5 ; PRINT I[2], ; NEXT I

```

The output of the line with label Example1 (shown below) is the numbers 1 through 5 printed one per line in a field width of two characters.

```

1
2
3
4
5

```

The output of the line with label Example2 (shown below) is the numbers 1 through 5 printed on a single line, each in a field width of two characters. Notice that the cursor would be left on the same line to the right of the 5.

```

1 2 3 4 5<cursor here>

```

F-Basic also supports the SPACE library function to aid in format control in PRINT statements. The syntax of SPACE is:

```

SPACE( <integer number of spaces> )

```

The following example uses SPACE to insert 20 blanks between the value of the two variables I and J:

```

PRINT I[3],SPACE(20),J

```

The value of I will be printed in the first 3 columns of the display, followed by 20 blanks, and the value of J printed in the next 12 columns.

Section 5-Cursor Control In F-Basic

F-Basic supports several graphics and text oriented high level features to aid the programmer wishing to create distinctive screen displays. Several of these features deal with the use of screens, windows, and graphics drawing on the Amiga, and are discussed fully in the chapter entitled 'High Level Graphics In F-Basic'. However, all programs may take advantage of the cursor control features discussed in this section to customize their output in a creative way.

The table below lists the various F-Basic high level statements that affect the cursor or screen display, the parameters expected, and a description of the resulting action taken during execution.

<u>-Basic Statement</u>	<u># Of Parameters</u>	<u>Action During Program Execution</u>
CURS_DN	1 Integer N	Moves Cursor Down N Lines
CURS_UP	1 Integer N	Moves Cursor Up N Lines
CURS_LE	1 Integer N	Moves Cursor Left N Characters
CURS_RI	1 Integer N	Moves Cursor Right N Characters
CURS_LOC	2 Integers M,N	Positions Cursor At Row M,Column N
CURS_POS	None	Reports The Current Position Of The Cursor To The User - See Below
CURS_INV	None	Cursor Is Not Displayed
CURS_VIS	None	Cursor Is Displayed
ERA_ALL	None	Screen Is Erased From Current Cursor Position To End Of Screen
ERA_EOL	None	Screen Is Erased From Cursor To The End Of The Current Line
ERA_LIN	None	Entire Line Containing Cursor Is Erased
INS_LIN	None	An Empty Line Is Inserted Directly Above Line Containing Cursor
SCR_BEEP	None	Produces Visual Screen 'Beep'
SCR_CLER	None	Clears Entire Screen / Homes Cursor
SCR_FORE	1 Integer N	Changes Foreground Color To Pen# N
SCR_BACK	1 Integer N	Changes Background Color To Pen# N
SCR_STYL	1 Integer N	Changes The Style Of Text Printed N=0 Will Print Plain Text N=1 Will Print Boldface Text N=3 Will Print Italic Text N=4 Will Print Underscored Text N=7 Will Print InverseVideo Text

So, for example, consider the following program segment:

```

SCR_CLER
CURS_RI(20)
DELAY(10)
CURS_DN(20)
DELAY(10)
CURS_LE(20)
DELAY(10)
CURS_UP(20)
DELAY(10)
CURS_LOC(10,10)
SCR_STYL(1)
PRINT "SCREEN BEEP",

```



```
SCR_BEEP  
DELAY(10)  
ERA_LIN  
SCR_STYL(0)
```

From the table above, we would expect the program segment to cause the display to be cleared and the cursor returned to the upper left corner when the program is executed. Then, the cursor will appear to move 20 columns to the right, 20 lines down, 20 columns to the left, and finally, 20 lines back up. Each of these program lines in the segment is followed by a one-second delay statement to enable the user to see the movement when the program is run. The next line causes the cursor to be positioned at row 10, column 10. The text 'SCREEN BEEP' appears at this position in the bold-face style, and the screen visually flashes at the user. After a short delay, the text on that line is erased, and the style is returned to plain text for future output to the screen.

As mentioned above, the function CURS_POS provides the programmer with a convenient way to obtain the current location of the cursor at any time. CURS_POS will return the current row and column position of the cursor in the upper and lower 16 bits, respectively, of the answer. For example, J=HIWORD(CURS_POS()) will return the current row position, while J=LOWORD(CURS_POS()) will return the current column position.

CURS_POS will return a -1 if it is executed at a time when only the AmigaDOS window is open, as it works specifically with 'user defined' windows. For more information on these types of windows, see the chapter entitled 'High Level Graphics In F-Basic'.

File Input And Output In F-Basic

Introduction:

F-Basic supports a variety of operations on sequential files. A sequential file is one in which the data items must be accessed in order starting with the beginning of the file. The following file operations are provided in F-Basic:

1. OPEN
2. OPENA
3. INPUT#
4. PRINT#
5. SKIPFILE
6. CLOSE

Section 1- Opening A File

To create a new file, or to access an existing file, the programmer must assign a unit number to the file name. Valid unit numbers are from 1 to 10. Either the OPEN or OPENA statement may be used. The syntax is:

```
OPEN( <unit number>,<string expression representing file name> )
```

The string expression may be a literal file name, a text variable whose value is the file name, or any string-valued expression which specifies the file name. For example:

```
OPEN(3,"datalist.ddd")
```

will create a file called 'datalist.ddd' on the current directory if no such file exists. Otherwise, it will open the existing file on the current directory in a mode that allows both reading from and writing to the file. The unit number associated with the file in either case is 3. The significance of the unit number will be apparent in later sections as the remaining file operations refer to the file by its unit number, rather than its file name.

If the file to be opened is not in the current directory it is necessary to specify a complete AmigaDOS pathname as part of the file

name. For example, if the file 'datalist.ddd' exists in the subdirectory 'sampprogs' of the current directory, one could open the file with:

```
OPEN(3,"sampprogs/datalist.ddd")
```

If the file exists on a drive other than the current drive, the file name path should include drive location information, as follows:

```
OPEN(3,"df2:sampprogs/datalist.ddd")
```

In this example, unit number 3 has been assigned to the file named 'datalist.ddd' residing in the directory 'sampprogs' on the second external drive.

Certain devices, other than actual files, may be treated as though they were disk files in F-Basic. Three of the most important are:

1. The Serial Port (RS-232 Port)
2. The Parallel Port (Printer Port)
3. The RAM Disk (A Virtual Disk Residing In Memory)

The following examples illustrate assigning a unit number to each of the above devices.

```
OPEN(1,"SER:")  
OPEN(2,"PAR:")  
OPEN(3,"RAM:datalist.ddd")
```

The last example creates or accesses a virtual file in memory named 'datalist.ddd'. After opening the serial or parallel device with this syntax, statements using the F-Basic PRINT# statement, as discussed below, will send the desired output to those devices as directed. To receive input from either of these devices, however, it is suggested that the programmer use direct calls to the AMIGA ROM library functions concerned with them.

When working with a sequential file, the operating system maintains a pointer to the next data item to be accessed in the file. The OPEN command discussed above initializes this system pointer to the beginning of the file. As will be discussed in Section 3, writing data to a file will place the data at the beginning of the file if the file existed when it was opened. This will cause the new data to overwrite the old data in the file. It is often times necessary, however, to append the data to the end of an existing file. This is facilitated by initializing the system pointer to the end of the file, rather than to the beginning. The OPENA command referred to above, has the same syntax as the OPEN statement and it opens the existing file

with the system's file pointer at the end of the file. In all other respects, it is exactly like the OPEN statement. There is no difference between the OPEN and OPENA commands if the file does not previously exist.

Section 2-Inputting Data From A File

To input data from a file, as opposed to from the keyboard, the programmer should use the INPUT# command. The syntax is:

INPUT# <unit number> <an optional transfer of control when the end of the file is encountered> <list of variables separated by commas>

For the sake of comparison, the keyboard input syntax is repeated below from the chapter entitled 'Keyboard Input And Screen Output In F-Basic'.

INPUT <list of variables separated by commas>

There are two obvious differences between INPUT and INPUT#. The first is that to INPUT# data from a file, one must specify the unit number associated with the file via an OPEN statement. INPUT, on the other hand, assumes that the input will come from the keyboard. The second difference is the presence of an optional statement that transfers control to another line of the program when the end of the file is encountered. A more detailed specification of the syntax of this statement is:

EOF= <statement number or statement label>

EOF stands for end of file. This option is very useful as the number of items to be read from a file is frequently not known in advance. When the end of the file being read from is encountered, any attempt to read further data items from the file will cause an error unless this optional EOF statement is specified. With the EOF statement present, the next statement executed is the one whose statement number or statement label is specified. The following program segment will open a file called 'Numbers' on the current directory. This file is assumed to contain an unspecified number of integers, one per line. The segment will read each of the numbers on the file, and print out their sum.

```
PROGRAM AddFromFile
INTEGER Sum,Number
OPEN(1,"Numbers")
```



```

      Sum=0
{NextNumber} INPUT #1 EOF=Print_Sum Number
      Sum=Sum+Number
      GOTO NextNumber
{Print_Sum}
      PRINT "The Sum Of The Integers In File Numbers Is",Sum
      END

```

In this program, unit number 1 is assigned to the file 'Numbers'. The file pointer is positioned at the beginning of the file automatically, and the variable Sum is initialized to 0. The INPUT# statement attempts to read the next item in the file. If the end of the file is not encountered the value of the integer from the file is stored in the variable Number, and control passes to the next statement where Sum is incremented appropriately. Control returns to the INPUT# statement because of the GOTO statement encountered next.

This process continues until the end of the file is reached. At this point, control passes to the statement labeled Print_Sum. No error message is generated by the F-Basic system during the execution of this program because an attempt to read past the end of the file did not occur. Although the EOF specification is optional in the INPUT# statement, it should be used unless the exact number of data items in the file is known in advance.

File input is in every other respect like the INPUT (from the keyboard) statement discussed previously.

Section 3-Printing To A File

Printing data to a file is accomplished in F-Basic with the PRINT# statement. The syntax is:

PRINT# <unit number> <list of expressions separated by commas>

The syntax of the PRINT# command is similar to the syntax of the PRINT command for printing data to the screen. The only difference is the requirement of a unit number to specify which file is to receive the data items. Particular care must be used when writing to a file which existed prior to being OPENed. If the file exists before the program is executed and the file is accessed using the OPEN statement, the file pointer will be initialized to the beginning of the file. All PRINT# statements that are executed will cause data to be written over the existing contents of the file. The old data items at the current position of the system file pointer will be lost. This occurrence is

sometimes desirable, but if the data is to be added to the file, as contrasted with replacing existing data, then the OPENA command discussed in Section 1 should be used.

In all other respects, using PRINT# to write data to a file is exactly like using the PRINT statement to write data to the SCREEN. For details regarding the formatted and unformatted writing of data items, see the chapter entitled 'Keyboard Input And Screen Output In F-Basic'.

To review, remember that the F-Basic system uses a SEPARATOR character, whose default value is the comma, to indicate where one data item ends and the next data item begins. The value of this character may be changed by using the &SEPARATOR command. For instance, the statement &SEPARATOR ; in a program will change the value of the SEPARATOR character to the semi-colon. Any use of PRINT# thereafter will separate data items on the same line with a semicolon in the file. It is sometimes desirable, however, to print data items to a file without a SEPARATOR character between the data items. This is particularly true if the file is to be later used as a document or report. The compiler directive &FILESEP 0 will suppress the SEPARATOR character on all PRINT# statements executed thereafter. The effect of this is to cause each line of the file to be treated as a single character string. To later read such a file from an F-Basic program, one must treat the file as having only one data item (a character string) per line. To restore the current value of the SEPARATOR character at any point in the program, the directive &FILESEP 1 should be used in the source program.

As with the PRINT command, if the list of expressions to be printed terminates with a comma, the next PRINT# statement will enter data into the file on the same line as the present PRINT# statement. In the absence of a trailing comma, each PRINT# statement creates a new line in the file. For an example program using this feature, refer back to the chapter which discusses the PRINT statement.

Section 4-Skipping To The End Of A File

When accessing a file using the OPEN statement, the file pointer is initialized to the beginning of the file. Any INPUT# or PRINT# statements then encountered will operate sequentially from that position in the file. If at some point in such a process it is desired to leave the present position in the file and begin adding data to the end of the file, the SKIPFILE command should be used. The syntax is:

SKIPFILE(<unit number>)

Section 5-Closing A File

In the multi-tasking operating system of the Amiga, any open file may not be accessed by another process or program. In particular, any file left open after program execution terminates will often be inaccessible by other programs until the system is rebooted. It is therefore good programming practice to use the CLOSE statement to close any open files before the program finishes execution. The syntax is:

CLOSE(<list of unit numbers separated by commas>)

For most F-Basic programs, the inclusion of this statement is not essential as the F-Basic system itself closes all open files when the program finishes execution.

The CLOSE statement is necessary in one circumstance. This occurs when a program wishes to access more than 10 files. Because the unit number is restricted from 1 to 10, a maximum of 10 files may be open and active at any one time. Before an eleventh file can be OPENed, the CLOSE statement must be used to free up a unit number for this new file.

After INPUT# or PRINT# operations have been performed, one may restore the system file pointer to the beginning of the file. This can be done by closing the file with the CLOSE statement, and then reopening the file with the OPEN statement.

Arrays In F-Basic

Section 1-One Dimensional Arrays

A one dimensional array is the structure in F-Basic that is used to represent lists of objects of the same data type. F-Basic supports arrays of any data type except PATTERN. The list as a whole is assigned a variable name called the array name. Individual objects in the list are referenced by specifying their integer position in the list. The position is also called the array index or array subscript. Arrays must be declared like other variable names in a declaration statement of the appropriate type. In addition, the array declaration must also specify the total number of items in the list. For example:

```
INTEGER Ages(30)
REAL Salaries(30)
TEXT*15 Names(50)
```

establishes Ages as a list of 30 integers, Salaries as a list of 30 real numbers, and Names as a list of 50 strings, each with maximum length 15. In the declaration of an array, the integer in parentheses is not a reference to a specific item in the list, but, instead specifies the total number of items in the list.

When referencing an array element, one specifies the array name followed by the index enclosed in parentheses:

```
<array name> ( <array index> )
```

Here, <array index> may be a literal, a variable, or more generally an expression. If the index is a variable or an expression, it must be an integer in the range 1 to the declared size of the array. A common source of logic errors in programs that use arrays is to use a variable or expression for the index whose value is out of this range. This error can't be diagnosed by the FastCom compiler; however, the Developer's Extension for F-Basic includes an option that allows run-time checking to verify that array indices are in range. If one has a program that uses arrays, and the program behaves irrationally or improperly, one should suspect that an array index is out of range. A simple way of diagnosing this problem is to conditionally print out the array index if its value is less than 1 or greater than the declared number of elements. Actually, for arrays of type INTEGER with literal subscripts, the F-Basic compiler performs automatic bounds checks.

Section 2-Array Equates

A variable name which is not declared as an array is referred to as a 'scalar'. A variable name which is declared as an array but is referenced without the accompanying subscript or index is a reference to the list as a whole. F-Basic supports two types of statements involving array names. The syntax of the first 'array equate' is:

`<array name> = <scalar>`

where `<array name>` and `<scalar>` have the same data type. Here, data type may be any of INTEGER, REAL, or TEXT. The right side of the equate, `<scalar>`, is evaluated, and the effect of this statement is to fill the entire list with its value. Referring to the examples of the previous section, the statement:

`Ages=100`

will fill all 30 entries in the array Ages with the value 100.

The second type of array equate has the syntax:

`<array name 1> = <array name 2>`

Here, the two array names must be of the same data type, but not necessarily of the same declared size. If N is the smaller of the two declared sizes, then the effect of the statement is to copy the first N entries from the list whose name is `<array name 2>` to the first N entries of the list whose name is `<array name 1>`.

Section 3-DATA Statements And Arrays

The use of DATA statements to initialize the value of scalar variables was discussed in the chapter entitled 'Variables And Data In F-Basic'. The reader unfamiliar with F-Basic's DATA statements is encouraged to consult that chapter before continuing. The use of DATA statements is a preferred way to initialize the values of an array as the DATA statement generates no executable code and thus, saves both time and space.

One technical remark is in order. In order to facilitate the implementation of DATA statements, the F-Basic system packages data storage in a single object program file along with the executable code

comprising the program itself. Thus, a large data storage declaration will increase the size of the executable program. Were this not the case, it wouldn't be possible to implement a DATA statement without slowing program development by requiring the use of a linking process.

DATA statements may be used to initialize all or part of any declared array. The syntax of a DATA statement involving an array is:

```
DATA ( <array name>,<list of values separated by commas> )
```

Here, a value may be any one of a literal value (including arbitrary base notation with the ' operator), a declared CONSTANT of the appropriate type, or a value with a specified number of repetitions. Referring again to the example of Section 1, one could initialize the value of Ages as follows:

```
DATA (Ages,20,15,30,10*5,6,22)
DATA (13,13,14,14,15,0*15)
```

This example illustrates many of the convenient features of the F-Basic DATA statement. An expression such as 10*5 in a DATA statement does not represent multiplication, but indicates instead that the value 10 is to be repeatedly assigned to the next 5 elements in the array. If one intended to assign the value 100 to all 30 elements of Ages, one could simply write DATA (Ages,100*30). A second feature illustrated above occurs when the number of values to be initialized is too large to fit conveniently on a single line. One can then use as many succeeding DATA lines as is necessary to assign values to all elements in the array. Notice that continuation DATA lines do not contain the name of the array, as that would indicate that the values in the DATA statement should be used to fill the array beginning with the first element.

DATA statements have a special feature when used to initialize arrays of type TEXT. When the text value contains fewer characters than the number of characters in an individual element, the remaining characters in the element are filled with blanks. Thus, to initialize the array Names so that each element contains all blanks, one need only state:

```
DATA (Names," "*50)
```

This is in contrast to the effect of using a variable of type TEXT in an assignment statement, where the variable is assigned fewer characters than its declared number of characters. In this instance, the remaining character positions in the TEXT variable are unaffected.

Alternatively, one may use a DATA statement to initialize a single array element. For example,


```
DATA (Names(5),"Jack")
```

will initialize the fifth element of the array Names to the literal string "Jack".

Section 4-The IN Operator And The SETARRAY Statement

An operation that occurs frequently in programs involving arrays is the process of locating the position of some data object in an array. For arrays of type INTEGER or type TEXT, F-Basic implements this operation in the form of the binary operator IN. The syntax for using IN is:

```
<scalar item> IN <array name>
```

Here, the left operand is a scalar of type INTEGER or TEXT, and the right operand is an array name of the same data type. The value of the operation is 0 if the scalar does not occur in the list and otherwise is an integer specifying the position of the scalar in the list. For example, if the scalar was the 10th element in the array, the operation would return the value 10. The following program initializes an array to the values 10,9,8...2,1. The program inputs an integer and prints out its position in the array or an appropriate error message if the integer is not in the array.

```
PROGRAM ArrayExamp
INTEGER I,K,Num(10)
DATA (Num,10,9,8,7,6,5,4,3,2,1)
INPUT "Enter an integer" K
I=K IN Num
IF I THEN
    PRINT "Your integer located at element #",I[2]," of Num"
ELSE
    PRINT "Your integer not located in Num"
ENDIF
END
```

Suppose that one enters the number 7 in response to the prompt. The IN operation will return the value 4 and the user will be so alerted. However, if 11 was entered at the prompt, IN will return the value 0 and the user will be informed that 11 is not in the array.

An important point to recall in connection with the use of IN to do a lookup in an array of type TEXT is that 2 text strings are equal only if they have the same length. Thus, the declared or literal length of the scalar must exactly match the declared character length

for each element in the array. For instance:

```
TEXT STR*20,ARRSTR(30)*15
INTEGER I
```

```
      .
      .
      I=STR IN ARRSTR
or I="GoodDay" IN ARRSTR
```

will always have result 0 for I, as the declared lengths of STR and ARRSTR are not the same, and the literal length of 'GoodDay' also does not match the declared character length of ARRSTR.

Sometimes an array may only partially be filled with meaningful data. The use of either the IN operator or array equates will reference all of the items in the list up to the declared length of the array. In order to speed up such operations, or to prevent erroneous results, F-Basic allows the effective length of the array to be reassigned to any value less than or equal to the declared size of the array. The SETARRAY statement should be used, and its syntax is:

```
SETARRAY ( <array name>,<array size> )
```

For example:

```
SETARRAY (Names,40)
```

would decrease the effective number of elements in the Names array to 40. Using the SETARRAY statement to set the value of an array to a size larger than the declared size does not generate a compiler error, as this must be noticed during run-time when the <array size> expression is evaluated, but will always result in undesirable and unpredictable behavior, including the overwriting of the program's data section.

Section 5-Multi-Dimensional Arrays in F-Basic

Many programming problems involve lists of data which are easier to conceptualize as geometrical arrangements of objects whose dimension is greater than one. For example, in a game of chess, the positions of the pieces are more naturally described by specifying the row and column of the piece on a chessboard rather than numbering the 64 positions in some sequential manner 1 to 64. Likewise, when representing the components of a three dimensional structure, it is more natural to give three coordinates to specify the position of some component. F-Basic supports arrays of any number of dimensions. For

example:

```
INTEGER ChessBoard(8,8)
REAL Position(30,40,50)
```

specifies ChessBoard as a two-dimensional array of type INTEGER, and Position as a three-dimensional array of type REAL. In the above examples, the integers that occur in the declaration statement in association with the array names specify the maximum value of the corresponding array index. For example, a reference to Position(I,J,K) will reference the REAL element in row I, column J, and layer K in the list Position. To be a proper array reference, I must be in the range 1 to 30, J in the range 1 to 40, and K in the range 1 to 50. As with singly subscripted arrays, a violation of this restriction is not detected by the FastCom compiler, but will result in undesirable and unpredictable behavior of the program. Again, use of the F-Basic Developer's Extension or judiciously placed PRINT statements will eliminate this type of problem.

The only constraint on the number of elements allowed in a singly-dimensioned array in an F-Basic program is the amount of memory available. For multi-dimensional arrays, the declared range of any subscript may not exceed 65536. This is not a serious limitation, as a declaration such as INTEGER A(70000,20) would require 5.6 megabytes of physical memory to store this list alone!

Subprograms In F-Basic

Introduction

When confronted with a complex programming problem, there are many advantages to dividing the problem into smaller subproblems and developing separate program units to handle each subproblem. First, it is easier to test and debug a small program unit than a larger unit. Secondly, it is easier to understand a program that is divided into smaller program units, each of which performs a limited and specified function. F-Basic supports the use of separate program units by supplying two types of subprograms called SUBROUTINE and FUNCTION.

Section 1-Overall Program Structure

Until now, this manual has considered only programs consisting of a single program unit, referred to henceforth as the 'main program'. The main program may optionally be accompanied by any number of subprograms. The structure of such a program is:

```
PROGRAM <name of main program>
    <declaration part of main program>
    <executable part of main program>
END
<Subprogram Header>
    <declaration part of subprogram>
    <executable part of subprogram>
END
<Second Subprogram Header>
    <declaration part of second subprogram>
    <executable part of second subprogram>
END
.
.
```

Here, <Subprogram Header> has one of two forms:

```
SUBROUTINE <name of subroutine>
or FUNCTION <name of function>
```

The program structure of a subprogram is like the program structure of the main program. One difference is that the main program begins with

the keyword PROGRAM, while a subprogram begins with either the keyword SUBROUTINE or the keyword FUNCTION. Other differences will be discussed in later sections of this chapter.

One new aspect of the <declaration part> of the main program is that it must contain a declaration of the names of the subprograms that follow it. The precise form of this declaration is also discussed in the appropriate sections of this chapter.

An additional new aspect of the <declaration part> of the main program is that it now may differentiate between variables that are 'LOCAL' to the main program, and those that are 'GLOBAL' to all program units. This too is fully explained in the sections which follow.

Section 2-Passing Information Between Program Units

There are only two ways in which distinct program units may share information. The first is by the use of GLOBAL variables, and the second is by the use of parameters. Except for GLOBAL variables and parameters, each program unit acts in complete isolation from any other program unit. A variable name which is not a GLOBAL variable or a parameter is called a LOCAL variable. One may use the same LOCAL variable names in different program units without any confusion. That is, a LOCAL variable named SUM in one program unit represents a different memory location, and thus a different value, from a LOCAL variable name SUM in a different program unit. One may reuse LOCAL variable names, statement labels, and statement numbers in different program units without causing any interaction or confusion. Apart from GLOBAL variables, each program unit uses symbols as if it were the only program unit in existence.

Section 3-GLOBAL Variables And The Declaration Part Of The Main Program

All program examples to this point in the manual have involved only LOCAL variables. This is because variables in the main program default to being LOCAL variables. A LOCAL variable is so named because its value can only be referenced within the program unit in which it is declared. A GLOBAL variable is so named because its value may be referenced from any program unit simply by using its variable name. GLOBAL variables can only be declared in the main program. GLOBAL

variables are initiated by the use of the statement GLOBAL. All declarations encountered after the GLOBAL statement, and until a LOCAL or SUBPROGRAM statement is encountered, will be treated by the compiler as GLOBAL variables. Thus, the declaration of GLOBAL variables must be the first statements in the <declaration part> of the main program, if GLOBAL variables are to be used. GLOBAL variables are not declared in any subprogram unit, but since subprogram units follow the main program in the program file, these variables may be referenced by any subprogram. The following example illustrates the use of both GLOBAL and LOCAL variable declarations in the main program unit:

```
PROGRAM FirstUnit
GLOBAL
    INTEGER First, Last, LIST(30)
    TEXT*50 Line
    REAL X, Y, Z
LOCAL
    INTEGER i, j, k
    TEXT*20 STR
SUBPROGRAM
    SUBROUTINE Sub
        INTEGER F
    .
    .
    .
```

The variables First, Last, Line, X, Y, Z, and the array LIST are all GLOBAL variables and may be referenced from any program unit. The variables i, j, k, and STR are LOCAL to the main program. If these latter names are re-used as LOCAL variables in any subprogram unit, they refer to different memory locations, and thus, bear no relation to the corresponding variables in the main program unit.

This sample also provides an example of the declaration of subprograms in the main program unit. The keyword SUBPROGRAM signals the beginning of the declaration of subprogram names. Recall that F-Basic supports two types of subprograms, SUBROUTINES and FUNCTIONS. The only difference between a subroutine and a function is that the name of a function is a variable name and has an associated value and an associated data type. The name of a subroutine is not a variable name, and thus, has no associated value or data type. Thus, subroutines are 'typed' by the keyword SUBROUTINE, while functions are 'typed' by the usual type statements. Therefore, in the above example, Sub represents the name of a subroutine, while F is the name of an integer valued function.

The rules regarding the <declaration part> of the main program may be summarized as follows:

1. GLOBAL variables are optional.
2. The beginning of GLOBAL declarations is signalled by the keyword GLOBAL.
3. In the presence of a GLOBAL statement, the beginning of LOCAL variables is signified by the keyword LOCAL.
4. In the absence of a GLOBAL statement, all variables default to LOCAL.
5. GLOBAL variable declaration statements must proceed all other variable declarations and are limited to the main program unit.
6. The keyword SUBPROGRAM signals the beginning of subprogram names.
7. Subroutine names are declared using the keyword SUBROUTINE, while functions are declared using their associated type declaration statement, be it INTEGER, REAL, TEXT, etc.

Section 4-Parameters And The Declaration Part Of Subprograms

The <declaration part> of a subprogram consists generally of two sections, PARAMETER declarations and LOCAL variable declarations. The form of the <declaration part> is:

```
PARAMETER
    <parameter variable declarations for the subprogram>
LOCAL
    <local variable declarations for the subprogram>
```

Subprograms are not required to have either PARAMETERS or LOCAL variables, but when both are present, the PARAMETER declarations must come before LOCAL variable declarations.

The difference between a PARAMETER and a LOCAL variable is that the value of a parameter is obtained from some other program unit at the time when control is passed to this subprogram, while the value of a LOCAL variable is obtained by computation or assignment from within the present program unit. The form of declaration statements in the LOCAL portion of the subprogram's <declaration part> is identical to declarations in the main program that have been discussed up to this point in the manual.

The form of declaration statements in the PARAMETER portion of the subprogram's <declaration part> are likewise identical to the declaration statements discussed up to this point, with two exceptions. First, the size of any PARAMETER array should be declared as *. Second, the length of any TEXT parameter should be declared as

\$. Consider the following example of a subroutine and its <declaration part>:

```
SUBROUTINE Sub
PARAMETER
    INTEGER I,J,K(*)
    REAL P,Q(*)
    TEXT L1*$,L2(*)*$
LOCAL
    INTEGER A,B(20)
    REAL C,B(3)
    TEXT STR1*5,STR2(15)*10
    .
    .
```

Notice that the declarations following the keyword LOCAL look similar to declarations encountered in previous program examples. On the other hand, in the PARAMETER declarations K(*), Q(*), L1*\$, and L2(*)*\$, the variables have either unspecified array sizes or in the case of the variables of type TEXT, an unspecified number of characters. Arrays and text variables passed as parameters to a subprogram acquire their values from corresponding arrays and text variables in other program units at the time when that program unit passes control to Sub. Since the size of these arrays may vary, and the length of the corresponding TEXT variables may vary, the size of the PARAMETER arrays and the length of PARAMETER TEXT variables need not be specified.

Section 5-Invoking Subprograms And Passing Parameters

To 'invoke' or pass control to a subprogram one simply references the name of the subprogram followed by a list of parameter values enclosed in parentheses. A FUNCTION subprogram may be invoked by using its name and accompanying parameters in any expression where a variable name with the same type as the FUNCTION would be appropriate. The syntax is:

<function name> (<list of parameter values separated by commas>)

Recall that a SUBROUTINE's name is not a variable name, and thus, its name may not be used in any expression. Therefore, to invoke a subroutine, one must write (as a separate line):

CALL <subroutine name> (<list of parameters separated by commas>)

The keyword CALL is optional. Note that passing control to a FUNCTION

may occur within some larger expression, while passing control to a SUBROUTINE constitutes a separate statement in F-Basic.

If MortgageInterest was declared to be a REAL-valued function with 2 REAL parameters, then the expression:

```
TotInt = TotInt+MortgageInterest(Principal,IntRate)
```

would transfer control to the FUNCTION MortgageInterest. Upon computing the value of MortgageInterest, the FUNCTION would return to this statement and finish the updating of the variable TotInt. In contrast, had MortgageInterest been declared to be a SUBROUTINE with 2 REAL parameters, control could be passed to MortgageInterest with the use of either of the statements:

```
MortgageInterest(Principal,IntRate)
or CALL MortgageInterest(Principal,IntRate)
```

With this arrangement, using the name MortgageInterest as though it were a variable name in an expression will produce a syntax error.

There are three rules which govern the use of parameter values. First, there must be exactly one value in the <list of parameters separated by commas> corresponding to each variable name in the PARAMETER declarations of the subprogram being invoked. Secondly, the order of the values listed (from left to right) must agree with the order in which the PARAMETER variables were declared (first to last). Finally, the type of each value in the <list of parameters separated by commas> must exactly match the type of the corresponding variable in the PARAMETER declaration. Thus, one may pass integer valued expressions to INTEGER parameters, real valued expressions to REAL parameters, etc.

Consider the following simple, but complete, example:

```
PROGRAM SubprogramExample
INTEGER N
REAL A(5)
DATA (A,1.5,1.0,2.5,3.0,7.0), (N,5)
SUBPROGRAM
    REAL SumArray
    PRINT SumArray(A,N)
END
FUNCTION SumArray
PARAMETER
    REAL B(*)
    INTEGER K
LOCAL
    INTEGER I
```



```

REAL Sum
Sum=0.
FOR I=1 TO K
    Sum=Sum+B(I)
NEXT I
SumArray=Sum
END

```

As SumArray is a FUNCTION of type REAL and not a SUBROUTINE, it is invoked here by referencing it in a PRINT statement as if it were an ordinary variable name. When the PRINT statement is encountered, control passes to the first statement of the <executable part> of FUNCTION SumArray. At the same time, the parameter array B and the parameter K acquire their corresponding values from the main program arrays A and variable N. Notice the first parameter value is the REAL array A. This corresponds to the first PARAMETER declared, namely the REAL array B. The second parameter value is the INTEGER scalar N, which corresponds to the second PARAMETER declared, the INTEGER K.

Unlike many programming languages, any mismatch between the number of parameter values and the number of PARAMETER variables declared, or between the type of a parameter value and the corresponding type of the PARAMETER variable declared will result in a compiler generated error.

Section 6-The Executable Part Of A Subprogram

With one exception, the executable part of a subprogram is identical to the executable part of the main programs discussed so far in this manual. In this section, that difference is fully explained.

In a main program, execution terminates when the END statement is encountered, or at any point at which a STOP statement is encountered. As the END statement must be the last statement of any program unit, one may think of the STOP statement as causing premature termination of the execution of the main program. By contrast, the END statement in a subprogram causes control to be passed back from the current subprogram to the statement which invoked it. Premature return of control from a subprogram can be accomplished by the use of the F-Basic GRETURN statement.

F-Basic supports three commands with similar names, but differing effects. The RETURN and LRETURN statements, discussed in the chapter entitled 'Control Structures In F-Basic', are used interchangeably to return control to a GOSUB statement located within the same program

unit. Recall that the GOSUB statement was used to create 'local' subprograms. Thus, GOSUB and RETURN or LRETURN define a control structure within the same program unit. Similarly, GRETURN is used in a subprogram of type SUBROUTINE or FUNCTION to return control to a different program unit; namely, the program unit that originally passed control to the current subprogram. Therefore, one may think of a subprogram of type SUBROUTINE or FUNCTION as a 'global' subprogram in that it may be invoked by any other program unit using the syntax described in the previous section.

Section 7-A More Detailed Discussion Of Passing Information Via Parameters

As noted above, F-Basic provides 2 mechanisms for transmitting information between different program units, GLOBAL variables and parameters. Each GLOBAL variable references a unique memory location. Thus, the use of a GLOBAL variable in the main program unit or any subprogram unit allows the programmer to access or modify the corresponding data value.

The mechanism of transferring information via parameters is somewhat more complex. F-Basic uses two methods of passing information through parameters which are commonly referred to as pass by value and pass by address. Pass by value is used exclusively with INTEGER, REAL, or pointer (discussed in a later chapter) scalar parameters. Parameters of any other type, such as INTEGER array, REAL array, TEXT, etc., use the mechanism of pass by address. The nature and significance of these two methods will now be explained.

In pass by value, the value of the expression used as a parameter is copied and assigned to the corresponding PARAMETER scalar upon entry into the subprogram. Any statement which modifies a parameter passed by value, therefore, modifies this copy and leaves the original value in the calling program unaltered.

In pass by address, the memory address of the expression used as a parameter in the calling program is assigned as the memory address of the corresponding PARAMETER variable in the subprogram. Any statement which modifies the value(s) of a parameter passed by address, therefore, modifies the corresponding original value(s) in the calling program.

Even though the distinction between these two mechanisms of parameter passing may be unfamiliar to the reader, their practical effect is easily understood. Pass by value, in effect, transfers information in only one direction-from the calling program to the

subprogram. Pass by address, on the other hand, allows information to be transferred in both directions. Thus, an INTEGER or REAL scalar parameter may not be used to directly return a value computed in the subprogram back to the main program. If one wishes to return an INTEGER or REAL scalar value computed in the subprogram back to the main program, any one of the following methods may be used:

1. Store the value in a GLOBAL variable.
2. Return the value by using a FUNCTION of type INTEGER or REAL.
3. Pass a pointer (the address of the location where the value is to stored) as a parameter to the subprogram.

This latter method is discussed fully in the chapter entitled 'RECORD Structures In F-Basic'.

Section 8-Recursion In F-Basic

Like most modern programming languages, F-Basic supports 'recursive' subprograms. This means that a subprogram may invoke itself. Recursion is especially useful in many complex programming problems. The following program illustrates recursion, as provided in F-Basic:

```
PROGRAM RecursExample
INTEGER N
SUBPROGRAM
    REAL Factorial
INPUT "Enter the Integer N" N
PRINT "The Factorial Of",N," Is",Factorial(N)
END
FUNCTION Factorial
PARAMETER
    INTEGER N
IF N<=1 THEN
    Factorial=1
ELSE
    Factorial=N*Factorial(N-1)
ENDIF
END
```

For example, Factorial(5) is defined as $5*4*3*2*1$, or 120. More generally, Factorial(N) is defined as $N*(N-1)*(N-2)*...*3*2*1$. The program line Factorial=N*Factorial(N-1) computes the Factorial of N by multiplying N and the Factorial of N-1. This is a simple instance of

recursion.

When a subprogram is entered, the F-Basic system automatically creates a new storage location in memory for each LOCAL variable in the subprogram. Unlike some language packages, this means that each level of recursion is accessing and modifying a different copy of the LOCAL variable. This is how you would expect any system claiming to support true recursion to function.

Using INCLUDE And APPEND To Combine Program Files

Introduction

Frequently, a program segment or program module may be used in several distinct applications. When this is the case, it is more efficient to place the frequently used code on a separate file and have each application that utilizes the code share the information from that file. This avoids the redundancy of typing the same program code into application after application, and enables a programmer to develop a convenient 'library' of frequently used routines.

Different programming languages utilize different mechanisms for accomplishing this goal. One way of implementing this capability is to allow separate modules of assembly or machine code to be linked to the parent application at compile time using the system linker. A second approach is to place the frequently used code segments, in the form of high level source statements, on a file that is accessed by the compiler during compilation of the parent application. These high level statements are inserted into the source code of the application at the location specified by the programmer. Paradoxically, when the speed of the two methods is compared in the case of the FastCom compiler, the second method proves to be much faster. As one of its most important features is its fast compilation times, the F-Basic language system relies entirely upon the second method of combining frequently used code. There are two statements in F-Basic which implement this feature, INCLUDE and APPEND.

Section 1-The INCLUDE Statement

The syntax of the INCLUDE statement is:

INCLUDE <filename>

Here, <filename> is the name of a file containing one or more lines of F-Basic source code to be inserted into the present program. One may think of the INCLUDE statement as being replaced by the lines of source code located in the file <filename>. If the file to be INCLUDED is not in the current directory it is necessary to specify a complete AmigaDOS pathname as part of <filename>. For example, if the file 'Declarations' is present on the current directory, then one may use:

INCLUDE Declarations

If, instead, the file 'Declarations' is contained in the directory 'MyLibProgs', the correct use of the INCLUDE statement would be:

```
INCLUDE MyLibProgs/Declarations
```

Unlike the <filename> which occurs in the F-Basic OPEN statement, discussed in the chapter entitled 'File Input And Output In F-Basic', the <filename> which appears in the syntax of the INCLUDE statement is not enclosed in double quotes or delimited. This is because the OPEN statement is executed at run time and permits either text variables or text literals as a specification of the file name. The INCLUDE statement, on the other hand, is evaluated at compile time. Therefore, variable references are not permissible in the INCLUDE statement.

The file INCLUDED may contain source statements comprising a portion of a module, or a complete module. Some of the statements on the INCLUDE file may themselves be INCLUDE statements. This 'nesting' of INCLUDE file references is permitted to a maximum depth of 8. The INCLUDE statement is typically used within the <declaration part> of a module to load complex declarations into the parent program. This is especially useful when defining RECORD structures, which are discussed in the chapter entitled 'RECORD Structures In F-Basic'. INCLUDE statements can also be used in the <executable part> of a module, such as graphics applications which involve opening custom windows and screens.

Section 2-The APPEND Statement

The syntax of the APPEND statement is:

```
APPEND <filename>
```

The APPEND statement acts exactly like the INCLUDE statement, except in one important respect. The contents of the file referenced with <filename> are inserted at the END of the current file, rather than at the place where the APPEND statement is encountered. Thus, one obvious use of the APPEND statement is to append entire subprograms that reside on other files to the parent application. The APPEND statement must belong to the <declaration part> of an F-Basic program. Currently, a maximum of 35 files may be queued for APPENDING.

Programming The Amiga's Sound Facilities

Introduction

This chapter presents a discussion of the many high level facilities provided by F-Basic to access the speech and sound capabilities of the Amiga. Naturally, these sound facilities may be invoked through direct calls to the Exec library using the SYSLIB interface provided in the language. However, for even easier access, F-Basic supports speech through three library functions: TRANSLATE, NARRATE, and VOICE. To produce more general sounds, F-Basic supplies four built-in routines: TONE, SOUND, WAVE, and WAVE_END.

Section 1-TRANSLATE, NARRATE, And VOICE

The F-Basic function TRANSLATE transforms a text string containing standard English words into a second text string containing the phonetic equivalent of this sentence. The actual translation is accomplished by the 'translator library' provided within the system libraries of the Amiga. The F-Basic function NARRATE uses this phonetic equivalent string to produce audible speech on the Amiga. In effect, it pronounces the string that resulted from the call to TRANSLATE. The syntax of the call to these routines is:

```
<rtnCode1>=TRANSLATE(<sourcestring>,<phoneme equivalent string>)  
and <rtnCode2>=NARRATE(<phoneme equivalent string>)
```

Here, a nonzero <rtnCode1> indicates that insufficient buffer space exists in the string <phoneme equivalent string> to hold the entire translation. In this case, the translation is broken at the end of the closest word in the sentence contained in the text variable <sourcestring>, and <rtnCode1> is set to the negative of the position in the sentence at which the translation ended. A non-zero <rtnCode2>, on the other hand, indicates that an error has occurred in the speaking process. A detailed list of error codes from the 'narrator.device' is provided in the Amiga ROM Kernel Manual.

The <phoneme equivalent string> is, of course, accessible to the user, and may be printed and/or modified. The Amiga's translator maintains a table of standard words which are custom translated into the correct set of phonemes. Words outside of the scope of this table are translated by an algorithm which may or may not produce the best

possible vocalization. The user may modify the <phoneme equivalent string> in such cases in order to provide a better speech quality. For hints and suggestions on how this may be accomplished, refer again to the Amiga ROM Kernel Manual.

The following sample program accepts an input string from the user and 'speaks' the sentence:

```

PROGRAM Speech
TEXT*120 Str1,Str2
INTEGER K
K=FILLCHAR(STR2," ")
    ? Note...REPEAT Forever
REPEAT
    INPUT "Enter Text To Speak Or <CR> To Terminate" Str1
    IF LENGTH(Str1)=0 THEN GOTO Finish
    ? Note...No Error Checking Done Here
    K=TRANSLATE(Str1,Str2)
    K=NARRATE(Str2)
    K=FILLCHAR(STR2," ")
UNTIL 0
{Finish} END

```

The F-Basic VOICE subroutine allows the user to alter the characteristics of the speech produced by the Amiga. The syntax of VOICE is:

VOICE(<word array name>)

Here, <word array name> represents the name of an array defined to be of type WORD, and containing at least six elements. The following table indicates which speech characteristics are encoded in each element of the array <word array name>:

Element #	Characteristic	Valid Range And Comments
1	Pitch	Specify Value Between 65 and 320 Hertz Default Is 110 (Male Speaking Voice)
2	Mode	A Value of 0 Means Normal Speech A Value of 1 Means Robotic Speech
3	Rate	Specify Value Between 40 and 400 Words Per Minute. Default Is 150
4	Sex	A Value of 0 Means Male Voice A Value of 1 Means Female Voice
5	Sampling Frequency	Specify Value Between 5000-28000 Hertz Higher Values Give Higher And Squeakier Voices. Default=22200
6	Volume	Specify Value Between 0 and 64

The following sample program is a modification of the previous program illustrating the use of the VOICE command. Note that it uses either a male or female voice.

```

PROGRAM Speech
TEXT*120 Str1,Str2
INTEGER K
WORD MaleV(6),FemaleV(6)
DATA (MaleV,110,0,150,0,22200,64),(FemaleV,300,0,125,1,22200,64)
INPUT "Enter 0 For Female Voice Or 1 For Male Voice" K
IF K THEN
    VOICE(MaleV)
ELSE
    VOICE(FemaleV)
ENDIF
K=FILLCHAR(STR2," ")
    ? Note...REPEAT Forever
REPEAT
    INPUT "Enter Text To Speak Or <CR> To Terminate" Str1
    IF LENGTH(Str1)=0 THEN GOTO Finish
    ? Note...No Error Checking Done Here
    K=TRANSLATE(Str1,Str2)
    K=NARRATE(Str2)
    K=FILLCHAR(STR2," ")
UNTIL 0
{Finish} END

```

NOTE: Programs which utilize NARRATE and TRANSLATE require the presence of the translator library and narrator driver software found on the Amiga boot disk. If they are not accessible on the default drive then both NARRATE and TRANSLATE will return error codes and not function.

Section 2-TONE

F-Basic provides the TONE function which produces a pure musical note at a specified frequency. The syntax is:

<integer>=TONE(<channel code>,<frequency>,<ticks>,<volume>)

The parameter <channel code> is an integer in the range 1-15. Bits 0 through 3 specify which of the four Amiga sound channels is to be

utilized. For a complete table linking channel assignment numbers to the left and right speaker jacks, refer to the Amiga ROM Kernel Manual.

The parameter <frequency> is an integer specifying the frequency of the tone in Hertz. Values may range from 30 Hertz to approximately 14,000 Hertz. For acceptable sound quality when playing very high or very low tones, external stereo quality speakers perform much better than the built-in speaker on the Amiga monitor. The parameter <ticks> is an integer specifying the duration of the tone in units of ticks (One tick is equal to 1/50 seconds). The last parameter, <volume>, is an integer specifying the volume of the tone. The volume may range from 0 to 255.

The function returns <integer> as a return code. If this is zero, the TONE function failed due to insufficient Chip memory or because the audio device provided on the boot disk was not available on the default drive during program execution. When successful, TONE returns a one.

For example:

```
K=TONE(7,523,500,100)
```

will produce a tone of frequency of 523 Hertz (approximately middle C) on both speakers for a duration of 10 seconds.

Section 3-WAVE, SOUND, And WAVE_END

The function TONE in the previous section generated a sound based on a sine wave. Sounds corresponding to more complex waveforms may be generated using the F-Basic routines WAVE, SOUND, and WAVE_END. To produce sound corresponding to a specified waveform, one must first store the waveform in an array of type BYTE. Next, a call to the function WAVE copies the waveform into Chip memory. The program may then execute one or more calls to the function SOUND to produce sound corresponding to the waveform. When the program is finished 'playing' the given waveform, the Chip memory must be deallocated with a call to WAVE_END. The syntax of the calls to these routines is:

```
<RtnCode1>=WAVE(<number of points>,<BYTE array name>)  
<RtnCode2>=SOUND(<channel code>,<frequency>,<ticks>,<volume>)  
<RtnCode3>=WAVE_END()
```

The parameter <number of points> specifies the number of points on the graph of the waveform. <BYTE array name> is the name of an

array of type BYTE whose values determine the amplitude of the wave. The value of <RtnCode1> is 1 if the routine was successful, and 0 if inadequate Chip memory exists during program execution.

The parameters to SOUND, and the resulting <RtnCode2>, are the same as the parameters and result for the TONE function, as discussed above.

Note that WAVE_END has no parameters. <RtnCode3> is set to 0 if a call to WAVE_END has been made in the absence of a previous call to WAVE. Otherwise, WAVE_END deallocates the memory and returns a 1.

The following sample program plays a one second saw-tooth waveform at a frequency specified by the user, and then replays the sound at twice that frequency:

```
PROGRAM SawTooth
BYTE WaveForm(16)
INTEGER Frequency,I
DATA (WaveForm,0,64,127,96,64,32,0,-32,-64,-96,-127,-64,0,0,0,0)
INPUT "Enter Frequency Desired" Frequency
? Note...No Error Checking
I=WAVE(16,WaveForm)
I=SOUND(7,Frequency,100,100)
DELAY(5)
I=SOUND(7,2*Frequency,100,100)
? Return ChipMem To System Free Memory Pool
I=WAVE_END()
END
```

The accompanying F-Basic sample programs disk contains more complex program examples using these routines.

NOTE: The product of the parameters <frequency> and <number of points> from SOUND and WAVE respectively must not exceed 29,886. If this value is exceeded, the Amiga hardware cannot produce the waveform at the desired frequency, and the resulting sound will be distorted.

High Level Graphics In F-Basic

Introduction

The Amiga has unquestionably the most powerful graphics of any personal computer available today in its price range. A programming language for the Amiga that fails to provide easy and convenient access to those facilities cannot and should not claim to be a superior general purpose Amiga language. More importantly, such a language does not provide the Amiga community with the type of commercial product that they have come to expect for their powerful machines.

The creators of F-Basic believe strongly in the graphics abilities of the Amiga. Therefore, they have taken great care to provide programmers with a multitude of ways to implement the graphical aspects of their applications. Those who customarily write programs in C will find that accessing the Exec, Graphics, and Intuition libraries of the Amiga with F-Basic is very similar. However, F-Basic provides high level statements for screens, windows, menus, and the mouse in a way that is useful to the C programmer, unlike most other BASIC packages. This ability to blend high level graphics statements with low level RECORD references and calls to Amiga library routines makes F-Basic a truly unique language. Additionally, the fast compile and execution times, the stand-alone executable code, and the ability to directly interface with the 68000 microprocessor, as provided in F-Basic, make it a developer's package with much to offer.

Those interested in producing Amiga graphics features through the use of RECORD structures and direct calls to the Amiga libraries are encouraged to also read the chapters entitled 'RECORD Structures In F-Basic' and 'Accessing Amiga ROM Libraries From F-Basic'.

This chapter discusses the high level statements for graphics output that are supported by F-Basic. Many are not usually available in other compiled BASIC packages. Therefore, extra attention to their purpose and use is provided. Some familiarity with the main ingredients in Amiga graphics programming, such as windows and screens, is assumed in the explanations.

Section 1-Custom Screens In F-Basic

The display characteristics of the Amiga are controlled through the use of a 'screen'. F-Basic allows a new screen to be created on the display with a single statement. The SCREEN command is used, and it has the following syntax:

```
<integer variable> = SCREEN #<screen integer> (<screen attributes>)
```

Here, the <screen integer> is an integer from 1 to 4. This number is used in other graphics statements to identify the screen. From this range of acceptable values, it is clear that F-Basic allows up to four custom or user-defined screens to be displayed at any one time. Using the statements described below, a screen may also be closed or taken off the display and reopened with new attributes any number of times. The screen number -1 is reserved for the WorkBench screen. This is the default screen displayed when the computer is first booted.

An opened screen always covers the full width of the display. The other characteristics of the screen are set by the contents of <screen attributes>. A more detailed syntax is:

TopEdge, Height, Depth, Mode, Title

where each of these is an integer expression with the following meaning:

TopEdge represents the position of the top of this screen.

Height represents the height of this screen.

Depth represents the number of bit planes to use for this screen.

Mode represents the type of this screen:

1 means low resolution, noninterlaced

2 means high resolution, noninterlaced

3 means low resolution, interlaced

4 means high resolution, interlaced

and, Title is a pointer to the string of characters representing the title to be displayed for this screen.

The TopEdge may be set to any value between 0 and 200 if a noninterlaced mode is chosen, or 0 and 400 if an interlaced mode is specified. A larger number brings the top of the screen down towards the bottom of the monitor. The Height chosen should be such that the sum of the TopEdge and Height does not exceed the limits just discussed. For example, a TopEdge of 20 and Height of 185 will put the last 5 rows of the screen off the bottom of the monitor if a noninterlaced mode is chosen. This will have undesirable and unpredictable results. Screens that are shorter than the full display height are acceptable and will cause the portion of the display below the screen to be filled with the background color.

As mentioned, Depth represents the number of bit planes to use

for the screen. This, in turn, determines the number of colors that are available in the screen at any one time. Valid Depths range from 1 to 5 for low resolution screens, and 1 to 4 for high resolution screens. The number of colors available is determined by raising 2 to the power Depth. Thus, a screen with Depth of 3 allows 8 different colors to be displayed at one time. As the value of Depth is increased, more memory is required to create and display the screen.

If the screen open operation is successful, F-Basic stores the address of the screen's data structure in the <integer variable> specified in the program line. If unsuccessful, a zero is returned. This address is useful if further direct calls to the Amiga library routines are made, as most require the address of the screen's RECORD structure.

As an example, consider the following two F-Basic lines:

```
ScreenPtr1=SCREEN #4 (0,200,3,1,0)
ScreenPtr2=SCREEN #1 (100,300,4,4,@"This is the Title Bar")
```

The first line will open a new screen whose identifying label is 4 provided that no other screen with identifier 4 is currently open, and also provided that sufficient CHIP memory is available for the screen. The screen will be low resolution and noninterlaced with a maximum of 8 colors available at any one time. Note that it will fill the entire display. It will not have a displayed title because the last parameter in the SCREEN statement is 0. The address of this Screen's data structure will be stored in the variable called ScreenPtr1.

The second line will open a new screen whose identifying label is 1 if the same conditions as explained above are true. The screen will be high resolution and interlaced with a maximum of 16 colors available. Additionally, it will occupy the exact middle of the display (200 units high out of a maximum 400 for a high resolution screen). When displayed, the screen will contain the title 'This is the Title Bar' at the top, and its address will be stored in the variable called ScreenPtr2.

Once a screen has been opened, F-Basic provides 3 statements that affect the screen's position in the display. They are:

```
SCREEN_CLOSE # <screen integer>
SCREEN_FRONT # <screen integer>
SCREEN_BACK # <screen integer>
```

The SCREEN_CLOSE statement will remove the screen from the display and free the memory used for it. The SCREEN_FRONT statement, when executed, will cause the specified screen to be displayed in front of all other screens, while the execution of the SCREEN_BACK statement causes the specified screen to be placed behind all other screens on

the display. The <screen integer> may be 1,2,3, or 4. Additionally, the SCREEN_FRONT and SCREEN_BACK statements accept -1 to indicate that the WorkBench screen should be moved as directed. If the indicated screen is not currently open, these statements have no effect.

Section 2-Windows In F-Basic

F-Basic provides several control statements that produce or change windows in the display screen. The most fundamental is the operation of creating a window in the display, which is directed with the WINDOW statement. The syntax is:

```
<integer variable> = WINDOW # <window integer> (<window attributes>)
```

Here, the <window integer> is an integer from 1 to 20. This number, like the screen integer above, is used in other graphics statements to identify the window. This range implies that F-Basic supports up to twenty user defined windows in the display at any one time, provided adequate memory exists. Using the statements described below, it is possible to close a window and reopen it with new attributes any number of times.

F-Basic allows significant flexibility in the choice of window parameters. The <window attributes> consists of 13 integer expression parameters which fully describe the desired window. A more detailed syntax is:

```
LeftEdge,TopEdge,Width,Height,MinWidth,MinHeight,MaxWidth,MaxHeight,  
DetailPen,BlockPen,Flags,Title,ScreenNum
```

The last parameter, ScreenNum, details which screen the window is to be opened in. From the first section of this chapter, the valid choices are 1 through 4 to indicate custom screens, and -1 to indicate the WorkBench screen. The choices for LeftEdge and TopEdge determine the coordinates of the upper left corner of the window relative to the chosen screen. Thus, a choice of LeftEdge=12 and TopEdge=10 places the upper left corner of the window 12 pixels to the right and 10 lines below the top left corner of the screen. The parameters Width and Height indicate the width and height of the new window, in these same units.

Great care should be taken to verify that the choices for the parameters LeftEdge, TopEdge, Width, and Height will keep the window within the boundaries of the screen in which the window is being placed. Failure to do so will cause undesirable and unpredictable results.

The values chosen for DetailPen and BlockPen determine the colors which are used to color the window. A value of -1 for either or both of these instructs F-Basic to use the same colors that were used in the screen designated by ScreenNum. (The default, therefore, is color registers 0 and 1 respectively for these parameters.) Positive values designate the various color registers available on the Amiga. For more explicit details, see the discussion of the COLOR_PENS and COLOR_DEFINE statements in Section 4 of this chapter.

The value of Flags indicates the requested attributes that the window will have. These include both system defined 'gadgets' and the type of refreshing that Intuition will use when the contents of the window are covered or it is resized. The following values are acceptable in the Flags parameter and result in these actions:

- 0 = SimpleRefresh Used With This Window
- 1 = Sizing Gadget Attached To Window
- 2 = Drag Bar Gadget Attached To Window
- 4 = Depth Arrangement Gadget Attached To Window
- 8 = Close Window Gadget Attached To Window
- 16 = SmartRefresh Used With This Window

Any or all of these options may be specified. To indicate more than one, simply add together the value of each feature desired. For example, to indicate that Drag Bar, Depth Arrangement, and Close Window Gadget, as well as SmartRefresh is desired, the Flags parameter should be set to 30 in the WINDOW statement.

Choosing SmartRefresh will enable Intuition to redraw the contents of the window after it has been covered or resized, but requires more memory. The SimpleRefresh option means that the portion of the window that is covered or resized is not remembered, and therefore, not redrawn when appropriate. SimpleRefresh requires less memory than its counterpart.

The values specified for MinWidth, MinHeight, MaxWidth, and MaxHeight are used only if a Sizing Gadget is chosen in the Flags variable. They indicate the smallest and largest the window is allowed to be changed to by the user through the use of this gadget. Again, failure to provide values that are sensible in lieu of the original size of the window will have undesirable and unpredictable results. If a Sizing Gadget is not chosen, these values are ignored.

Finally, Title represents the address of the string of characters to be displayed in the title bar of this window. A value of 0 indicates no title should be included in the window.

If the window number given to this window is not already opened, if sufficient CHIP memory is available, and if the screen specified is

currently open, the WINDOW statement will place the new window on the display and the address of its data structure in <integer variable>. If the window cannot be opened, 0 will be stored in the variable.

Recall the example SCREEN statements from the first section of this chapter. If the following two F-Basic lines were also included:

```
W1=WINDOW #20 (0,0,640,200,0,0,0,0,-1,-1,0,0,-1)
W2=WINDOW #1 (0,0,300,150,20,20,350,180,1,2,31,"Window 1",1)
```

two windows would be produced, provided the conditions explained above were met. The first window, identified by the label 20, would appear on the WorkBench screen. Since this screen is a high resolution noninterlaced one, notice that window 20 would fill the entire screen. It will have no gadgets attached, and no title displayed. Its colors are set to the default coloring used in the WorkBench screen. W1 will hold the address of this window's RECORD structure, as returned by Intuition.

The second window will be displayed in the high resolution interlaced screen labeled 1, as defined and opened above. Its definition sets the top left corner of the window at position (0,0), which is the top left position available in screen 1. The window will be 300 wide by 150 high. All system gadgets and Smart Refresh have been selected. Because the sizing gadget is included, the values designated in parameters 5 through 8 indicate that the user may resize the window to as small as 20 by 20, and as large as 350 by 180. The color defined to exist in color register #1 will be used for the DetailPen of this window, and that of register #2 will be used to produce the BlockPen features of the window. Finally, this window will have the title 'Window 1' displayed in its title bar.

Once a window has been opened, F-Basic provides 7 statements that affect the window's position in the display. Listed in alphabetical order, they are:

```
WINDOW_ACTIVE # <window integer>
WINDOW_BACK   # <window integer>
WINDOW_CLOSE  # <window integer>
WINDOW_FRONT  # <window integer>
WINDOW_OUTPUT # <window integer>
WINDOW_SIZE   # <window integer> (DX,DY)
WINDOW_MOVE   # <window integer> (DX,DY)
```

Each of these operations is without effect if the window selected by <window integer> is not currently open. WINDOW_FRONT brings the designated window to the front of all other windows currently open on its screen. WINDOW_BACK places the designated window to the back of all other windows currently open on its screen. WINDOW_CLOSE closes the designated window freeing its resources for use by later

operations requiring them.

Those familiar with the Amiga window interface are well aware that only one window may be 'active' at any one time. The active window is the one that is fully featured on the display, while the inactive windows are ghosted or drawn in silhouetted colors. When a window is opened with the F-Basic WINDOW statement, it automatically becomes the active one, and all others are inactivated. Additionally, this new window automatically becomes the current output window in the F-Basic system. This means that the output from any PRINT, INPUT, or COLOR graphics statements used in the program after opening the window are automatically directed to this new window. A window can accept and draw text and graphics output, regardless of whether it is active or inactive. The active window, however, is the only one with which the user may interact using the Intuition interface.

The user is free at all times to manipulate the mouse and make another window the active one simply by clicking on it. Additionally, it is likely that if a program opens multiple windows, the programmer may wish to redirect the output of the program to a window other than that which was most recently opened. The WINDOW_ACTIVE and WINDOW_OUTPUT statements are closely related, and are designed to give the F-Basic programmer full control over the window interface.

The WINDOW_ACTIVE statement, when executed, will force the designated window to immediately become the active one on the display, if that window is currently open. This should be used with care, as deactivating a window that the user is currently interacting with may appear to be somewhat rude. The WINDOW_OUTPUT statement tells the F-Basic system to direct all future text and graphics output, as well as prompts from the INPUT statement, to the window designated with <window integer>. The WINDOW_OUTPUT statement also accepts the integer 0 as the <window integer> as a special case to indicate that text output should be redirected back to the opening CLI window from which the F-Basic program was invoked.

The WINDOW_SIZE and WINDOW_MOVE statements resize or move the window as directed by the program. The parameters DX and DY may be positive or negative integer expressions that represent the change to be made in the designated window's size or position on the screen. For example, the statements:

```
WINDOW_SIZE #1 (10,-12)
WINDOW_MOVE #1 (5,5)
```

will resize window 1 by expanding its width 10 units, and reducing its height by 12 units. The window's position on the screen will also be changed by moving it 5 units to the right and 5 units down. Again, it is extremely important to verify that changes such as these to the window do not place it outside of the bounds of the screen containing

it. This typically will have undesirable and unpredictable results.

The user is able to change the window's size and/or position after it is opened (assuming a sizing and/or drag bar gadget were specified in the Flag parameter), and therefore, the programmer must have a way of determining the current position, size, and other details of the window at any given time. F-Basic provides the WINDOW_INFO function to obtain current information about the state of the display. The syntax is:

`<integer result>=WINDOW_INFO(<window info integer>)`

The value specified for the <window info integer> can be 1 through 9, and it determines the value returned by the function. A complete description of the possible values returned by the WINDOW_INFO function is:

<u>Parameter</u>	<u>Value Returned By The WINDOW_INFO Function</u>
1	Decimal Number of the Current Output Window (1-20)
2	Address of the Current Output Window's Data Structure
3	Address of the Current Output Window's Rastport Structure
4	Width of the Current Output Window
5	Height of the Current Output Window
6	X-Coordinate Location of the Drawing Pen in the Current Output Window
7	Y-Coordinate Location of the Drawing Pen in the Current Output Window
8	TopEdge Location of the Current Output Window
9	LeftEdge Location of the Current Output Window
10	X-Coordinate of the Mouse In The Current Output Window
11	Y-Coordinate of the Mouse In The Current Output Window

Section 3-Text In F-Basic Windows

Each window opened with the F-Basic WINDOW statement has its own console device attached to it. Thus, each window has its own visible cursor. All of the keyboard and screen operations available with the default CLI environment of the F-Basic system are also available to the programmer who has opened a custom window. These include the PRINT, INPUT, CURS, and SCR statements. The conveniences of formatted output, program controlled input from the keyboard, direct line and blank control, and direct cursor control are available in any custom window opened with the F-Basic graphics system. The correct syntax and use of these statements is discussed in the chapter entitled 'Keyboard Input And Screen Output In F-Basic'.

F-Basic also allows direct access to the text-oriented functions in the Graphics library of the Amiga's Kernel. Programmers desiring the pinpoint control of text graphics unavailable with the console device can conveniently continue to use the Amiga's libraries from F-Basic. See the chapter entitled 'Accessing Amiga ROM Libraries From F-Basic' for further details.

Section 4-Colors, Points, Lines, And Drawing In F-Basic Windows

The WINDOW and SCREEN statements discussed in Sections 1 and 2 allow the programmer great flexibility in opening custom screens and windows in an Amiga application. These functions purposely return the addresses of the WINDOW and SCREEN data structures to enable the direct use of the data structure in all of the library functions provided by the Amiga's Kernel. The C programmer, for example, is already used to the direct use of these addresses in calls to library routines that manipulate graphics and text, among other things. F-Basic provides high level access to every one of the Amiga library routines, and, inasmuch, grants the ability to support any application that is currently written in C. The discussion of Amiga library routines can be found in the chapter entitled 'Accessing Amiga ROM Libraries From F-Basic'.

However, many F-Basic users are not that anxious to interact directly with the Exec, Graphics, or Intuition libraries of the Amiga. Therefore, several high level graphics statements are supported in the language. Many graphics intensive applications can be written in F-Basic simply by using these functions, thereby avoiding the need to understand the complexities of the operating system. All programmers, whether novice or experienced, will benefit from the ease of combining both high level statements and low level library calls in the convenient way provided by F-Basic.

The 32 color registers of the Amiga may be directly set through the use of the COLOR_DEFINE statement in F-Basic. The syntax is:

COLOR_DEFINE #<color register integer> (Red,Green,Blue)

In all COLOR graphics statements, the <color register integer> is an integer from 0-31 specifying which register the operation affects or uses. The values of Red, Green, and Blue are integer expressions representing the appropriate level of that color to be used in defining the contents of the color register specified with <color register integer>. The valid range is from 0 to 15. For example, the lines:


```
COLOR_DEFINE #2 (15,0,0)
COLOR_DEFINE #4 (15,8,0)
COLOR_DEFINE #5 (15,15,15)
```

define color register 2 to contain the color red, register 4 to contain the color orange, and register 5 to contain white. Any graphics statements executed that specify these registers will draw images with these colors. As each of the three colors can be set to one of 16 levels, there are 4096 different colors available. Remember that the number of bit planes defined in the SCREEN statement limit the number of colors that may be displayed on the display at any one time. A complete list of the default colors of the 32 registers is provided in the appendix entitled 'The Default Colors Of The 32 Amiga Color Registers'.

F-Basic uses the foreground and background pens of the Amiga to display graphics and text. The programmer may change the colors of these two drawing pens through the use of the COLOR_PENS statement. The syntax is:

```
COLOR_PENS (<foreground pen>,<background pen>)
```

Here, <foreground pen> and <background pen> are integer expressions whose value is in the range 0-31. They specify the color register, and thus the color, of the two drawing pens. Given the COLOR_DEFINE definitions above, this statement:

```
COLOR_PENS (2,4)
```

would set the foreground pen to draw in red, and the background pen to draw in orange.

The simplest graphics operation possible consists of coloring a single pixel in the current output window. This is accomplished with the F-Basic COLOR_POINT statement, whose syntax is:

```
COLOR_POINT #<color register integer> (X,Y)
or COLOR_RPOINT #<color register integer> (X,Y)
```

In the COLOR_POINT statement, the pixel to be colored is located X pixels to the right and Y pixels below the top left corner of the window, which is noted as position (0,0). Both X and Y should be positive. In the COLOR_RPOINT statement, the pixel to be colored is referenced relative to the current location of the foreground drawing pen. Thus, X and Y represent displacements from that location and may be either positive or negative. The 'R' in graphics statements will be used frequently to indicate the 'relative' mode is used to calculate the actual location on the display. Again, the reader is cautioned that graphics operations should be kept inside the bounds of the window to avoid undesirable and unpredictable results.

The result of either of these statements is to color the pixel with the color contained in the register whose number is given by <color register integer>, and to move the drawing pen to the point colored. The #<color register integer> is actually optional. If not part of the statement, the pixel is colored with the current color of the foreground pen. Thus, the statements:

```
COLOR_POINT #2 (20,30)
COLOR_RPOINT (10,10)
```

will color the pixel at location (20,30) in the current output window with the contents of color register 2, and then color the pixel at location (30,40) with the contents of the current foreground pen color.

Closely related to coloring a pixel is the operation of reading the current color of a pixel. The COLOR_READ function is provided, and its syntax is:

```
<integer variable> = COLOR_READ (X,Y)
or <integer variable> = COLOR_RREAD (X,Y)
```

The COLOR_READ statement will return the number of the color register of the pixel at location (X,Y), and store it in the <integer variable>. The COLOR_RREAD statement functions the same, except that it reads the pixel that is located X pixels horizontally and Y pixels vertically from the current location of the drawing pen in the window (relative mode).

The syntax of the F-Basic statements used to draw lines, boxes, and filled boxes is closely related. For example, the syntax of the COLOR_LINE statement is:

```
COLOR_LINE #<color register integer> (<ModeFlag>,X1,Y1,X2,Y2)
```

In this statement, the line will be drawn in the color contained in the register designated by <color register integer>. As with the COLOR_POINT statement, the #<color register integer> is optional. If not present, the line is drawn with the current foreground pen. The <ModeFlag> tells the F-Basic graphics system how to interpret the values of the remaining 4 parameters. The possible values and interpretations are:

<u><ModeFlag> Value</u>	<u>Meaning</u>
0	1st Coordinate Absolute, 2nd Coordinate Absolute
1	1st Coordinate Relative, 2nd Coordinate Absolute
2	1st Coordinate Absolute, 2nd Coordinate Relative
3	1st Coordinate Relative, 2nd Coordinate Relative

The pixel location (X1,Y1) represents the beginning of the line if the <ModeFlag> is chosen to represent the first coordinate as absolute. If relative is chosen instead, X1 and Y1 indicate the displacements from the current location of the drawing pen to be used as the beginning of the line. The end of the line is represented by (X2,Y2), in either an absolute or relative sense, again as determined by <ModeFlag>. The drawing pen is moved to the end of the line upon completion of the COLOR_LINE statement.

The COLOR_BOX and COLOR_BOXFILL statements are used to draw a box or a filled box respectively in the current output window. The syntax of these statements is exactly the same as the COLOR_LINE directive, including the optional pen number and 5 parameters. Here, though, the location (X1,Y1) represents the upper left corner of the box, and (X2,Y2) represents the lower right corner of the box. Both locations are either in an absolute or relative sense, as determined by <ModeFlag>. Upon completion of the COLOR_BOX or COLOR_BOXFILL statement, the drawing pen is positioned at the upper left corner of the box. For example:

```
COLOR_BOXFILL #3 (0,10,20,50,60)
```

will produce a box in the current output window that will be filled with the color contained in register 3. Its upper left corner will be at pixel location (10,20) and its lower right corner will be at pixel location (50,60). Thus, this box will be 40 pixels wide and 40 pixels high.

Circles and ellipses can be drawn using the F-Basic COLOR_ELLIPSE statement. The syntax is:

```
COLOR_ELLIPSE #<color register integer> (Xcen,Ycen,Xrad,Yrad)
or COLOR_RELLIPSE #<color register integer> (Xcen,Ycen,Xrad,Yrad)
```

As before, the #<color register integer> is optional, and if present, indicates the color used to draw the figure. If not present, the current foreground color is used. The pixel location (Xcen,Ycen) indicates the absolute location for the center of the figure if the COLOR_ELLIPSE statement is used. If COLOR_RELLIPSE is specified, the values of Xcen and Ycen are added to the current location of the drawing pen to determine the location of the center. In both cases, the drawing pen is moved to the exact center of the figure. Finally, Xrad and Yrad indicate the horizontal and vertical radii to be used to draw the ellipse. For instance:

```
COLOR_ELLIPSE (90,100,30,40)
```

will produce an ellipse on the current output window whose center is at pixel location (90,100), whose horizontal radius is 30 pixels, and whose vertical radius is 40 pixels. Naturally, by refining the values

of Xrad and Yrad to take into account the resolution of the current output window, a perfect circle can be formed with the COLOR_ELLIPSE statement.

F-Basic supports the drawing of general polygons through the use of the COLOR_POLYDRAW statement. The syntax is:

COLOR_POLYDRAW #<color register integer> (<ArrayAddress>,<NumCoordPairs>)

Again, the #<color register integer> is optional, and if present, specifies the color register to use when drawing the polygon. If omitted, the current foreground color is used. The parameter <ArrayAddress> is used to specify the address of an array of type WORD that has been filled with the pixel locations of the polygon corners. As shown in the example below, the first entry in the array is the X location of the first corner of the polygon, followed by the Y location of this point. The array's third entry is the X location of the second corner, followed by the Y location of this point. This process is continued until all of the polygon's corners are specified. Notice that the polygon is automatically connected between the last and first coordinate pair specified, without explicitly including the first point in the array twice. The parameter <NumCoordPairs> indicates the number of coordinate pairs, and should equal the number of entries in the array divided by 2.

The following example displays an octagon in a window opened on the WorkBench screen:

```
PROGRAM Octagon
INTEGER WindowPtr
WORD Coords(16)
DATA (Coords,180,50,210,80,210,120,180,150,100,150)
DATA (70,120,70,80,100,50)
WindowPtr=WINDOW #1(20,20,600,160,0,0,0,0,-1,-1,0,0,-1)
COLOR_POINT(100,50)
COLOR_POLYDRAW #5 (@Coords,8)
DELAY(20) ; ? delay for a bit to show off
WINDOW_CLOSE #1
END
```

Notice that the drawing pen was moved to location (100,50) with a COLOR_POINT statement to avoid an extraneous line from the original location of the drawing pen to the first corner of the octagon.

Section 5-Patterned Graphics In F-Basic

Those familiar with the graphics support provided for the Amiga are well aware that its Kernel routines use 'bit-patterns' to draw lines, fill boxes with colors, and to fill geometric shapes with colors. Different effects can be achieved simply by changing the bit-patterns used by the graphics routines. F-Basic supports several high level features that take direct advantage of this capability, and they are the subject of this section. Those unfamiliar with the Amiga's use of bit-patterns are first encouraged to consult the ROM Kernel Manual or one of the many Amiga Graphics books currently available.

The most fundamental graphics pattern operation involves changing the bit-patterns used by the system. F-Basic provides the COLOR_PATTERN statement for this task, and its syntax is:

```
COLOR_PATTERN (LinePattern,AddressAreaPatternArray,Log2NumElem,Mode)
```

This one statement allows the programmer to control each of the fundamental drawing patterns. LinePattern is an integer valued expression whose lower 16 bits represent the new dotted, solid, or combination line pattern used by the Amiga graphics hardware for all future line drawing operations. For example, FFFF'16 represents a solid line, while AAAA'16 represents a uniformly dotted line.

The address specified in the location for the second parameter, AddressAreaPatternArray, represents the memory location of the start of an array declared to be of type WORD that contains the binary representation of the area pattern. This is used by the Amiga's graphics hardware when rendering filled shapes into the display. Accordingly, the third parameter, Log2NumElem, is an integer expression representing the logarithm base 2 of the number of elements placed in that array.

Finally, the Mode parameter represents the drawing mode used by the hardware when placing new images over existing ones. Again, the ROM Kernel manual discusses the drawing modes of the Amiga extensively. In F-Basic, the values for Mode correspond to the following drawing modes:

<u>Value of Mode</u>	<u>Amiga Drawing Mode</u>
0	JAM1
1	JAM2
2	COMPLEMENT
4	INVERSEVID

The values given above may be added together to produce a Mode parameter that requests more than one drawing mode, as is shown next. The example line:

```
COLOR_PATTERN (AAFF'16,@AreaPAT,4,5)
```


will change the Amiga's line pattern to AAFF'16, or equivalently 1010101011111111'2. Additionally, the Amiga's area pattern will be set to the contents of the WORD array AreaPAT, which apparently has 2⁴ or 16 elements. Finally, the drawing mode will be changed to JAM2 and INVERSEVID.

The COLOR_PATTERN statement also accepts the normal values used to indicate multi-colored patterned area fills, as explained in the ROM Kernel manual. Thus, negative values for Log2ElemNum are accepted by F-Basic without hesitation.

If the value of AddressAreaPatternArray is -1, then no change to the area pattern of the Amiga is made, and the value of Log2NumElem is ignored. Likewise, if the value of the Mode parameter is -1, no change is made to the current drawing mode.

The F-Basic statement COLOR_FLOOD is used to fill an area with a specified color using the current state of the Amiga's bit-patterns as defined above. Thus, solid or patterned fills are possible using the COLOR_FLOOD statement. Its syntax is:

```
COLOR_FLOOD #<color register integer> (X,Y,BorderPen)
or COLOR_RFLOOD #<color register integer> (X,Y,BorderPen)
```

Here, as before, the #<color register integer> is optional, and if present, indicates the color that should be used to 'flood' the figure with. If not present, the current foreground color is used.

The parameters X and Y indicate the absolute pixel coordinates at which the color flood should begin if the COLOR_FLOOD statement is used. If, instead, the COLOR_RFLOOD statement is used, X and Y are added to the current location of the drawing pen to determine the pixel location at which the flood is to begin.

Using the FLOOD commands demands special attention to the details of the current window. This is because the FLOOD commands continue to fill the window in all directions with the specified color until a line drawn with the pen color specified in the parameter BorderPen is encountered. This is known as the Amiga 'outline method of filling' with color. If the shape that is chosen to be FLOODEd is not completely enclosed by the BorderPen color, the FLOOD color will escape through the opening and spread out to cover the entire window. Likewise, if a border in the BorderPen color is not found, the FLOOD of color will fill the entire window. If the BorderPen parameter is set to -1 in the call to the COLOR_FLOOD routine, then the current foreground color pen is taken as the color of the enclosing border.

The last of the pattern-fill operations supported by F-Basic allows the programmer to specify a polygon that is to be filled with

the colors and patterns dictated by the present state of the Amiga's bit-patterns. This is accomplished by specifying a list of points on the screen to be connected with lines, and filled with patterned colors. The F-Basic COLOR_AREA, COLOR_RAREA, and COLOR_AREAFILL commands control the operation. Their syntax is:

```
COLOR_AREA (X,Y)
COLOR_RAREA (X,Y)
COLOR_AREAFILL
```

The parameters X and Y indicate the absolute pixel coordinates of the next point to be included as a corner of the polygon if the COLOR_AREA statement was used. If the COLOR_RAREA command is used instead, X and Y indicate the next corner's pixel location relative to the current location of the drawing pen. In both cases, the drawing pen is moved to the point so described.

The actual drawing and filling is not done until the COLOR_AREAFILL statement is encountered. This command instructs the F-Basic system to complete the current shape as defined by all previous AREA statements. Note that a maximum of 30 points in the polygon may be defined (one at a time) using the F-Basic COLOR_AREA or COLOR_RAREA statements. Any additional points above this limit are ignored by the system.

Section 6-Manipulating Image Blocks In F-Basic

It is sometimes desirable to translate portions of graphical objects within a window. The F-Basic functions COLOR_SCROLL and COLOR_RSCROLL permit the user program to translate all objects within a specified rectangle by a given displacement DX horizontally and DY vertically. The contents of the window outside this rectangle are not affected. In this context, object refers to any region drawn in a color other than the background color. Thus, all portions of the rectangle except the background color are relocated.

The syntax is:

```
COLOR_SCROLL (X1,Y1,X2,Y2,DX,DY)
or COLOR_RSCROLL (X1,Y1,X2,Y2,DX,DY)
```

where, for COLOR_SCROLL, (X1,Y1) is the pixel coordinate of the upper left hand corner of the rectangle, and (X2,Y2) is the pixel coordinate of its lower right hand corner. For COLOR_RSCROLL, (X1,Y1) is the pixel coordinate of the upper left hand corner of the rectangle relative to the current position of the drawing pen, X2 is the

horizontal width of the rectangle, and Y2 is its vertical height. In both functions, DX and DY represent the distance that non-background objects are to be moved within the boundary of the specified rectangle. A positive DX causes motion to the right, while negative DX causes motion to the left. Likewise, a positive DY causes motion downward, while a negative DY causes upward scrolling. (NOTE: This convention for DX and DY is consistent with the specification of the upper left hand corner of a window as pixel (0,0), although the equivalent ROM Kernel function uses the opposite sign convention for DX and DY.)

F-Basic also allows the graphical contents of a specified rectangle within a given window to be saved in a separate portion of CHIP memory. Later, the contents of this 'saved' rectangle may be copied to a different position within the same window or within a different window. The saved rectangle is referred to as a BLOCK. F-Basic permits up to 20 BLOCKs to be stored within CHIP memory at any time. The syntax of the command to save a rectangle of data is:

```
<error code>=BLOCK_GET #<block number> (<ModeFlag>,X1,Y1,X2,Y2)
```

BLOCK_GET allocates space in CHIP memory and copies the specified rectangle from the current output window into that space. <block number> is a literal integer in the range 1 to 20, and is used in any subsequent reference to this saved rectangle, as discussed below. The coordinate (X1,Y1) specifies the upper left hand corner of the rectangle, while (X2,Y2) specifies its lower right hand corner. Each (X,Y) pair may either represent absolute or relative coordinates. The parameter <ModeFlag> determines which of the four possible combinations is in force. The manner in which <ModeFlag> should be chosen was discussed in Section 4 of this chapter in connection with the COLOR_LINE, COLOR_BOX, and COLOR_BOXFILL functions. BLOCK_GET returns a zero in <error code> if the call fails due to insufficient CHIP memory or the non-existence of a current output window. Otherwise, upon successfully saving the rectangle, it returns one.

Once a rectangle has been copied into a BLOCK of CHIP memory, the contents of that BLOCK may then be transferred any number of times to any available window by using either of the following commands:

```
<error code>=BLOCK_PUT #<block number> (X1,Y1,<Minterm>)
<error code>=BLOCK_RPUT #<block number> (X1,Y1,<Minterm>)
```

Here, <block number> again represents a literal integer in the range 1 to 20 corresponding to the value used by BLOCK_GET in saving the BLOCK. For BLOCK_PUT, (X1,Y1) represents the absolute pixel coordinate of the upper left corner of the destination rectangle in the current output window. For BLOCK_RPUT, these coordinates are computed relative to the current pixel location of the drawing pen. To enable a BLOCK to be copied from one window to a different window, it

is only necessary to insert a call to WINDOW_OUTPUT, as explained in section 2 of this chapter, between invoking BLOCK_GET and BLOCK_PUT.

An <error code> of zero upon return from a BLOCK_PUT call indicates a failure of the put operation because either the specified BLOCK had not been created with a previous BLOCK_GET statement, or a current output window was not available. An <error code> of one indicates a successful put operation.

The function of <MinTerm> is somewhat complicated. If its value is C0'16 (192 base 10), both BLOCK_PUT and BLOCK_RPUT simply copy the contents of the saved rectangle into the specified destination rectangle. The BLOCK_PUT functions are actually more flexible than this, as they can replace the contents of the destination rectangle with any boolean function of the saved block and the previous contents of the destination rectangle. The pixel code of any pixel in the saved BLOCK will be represented by the variable B. The pixel code of any pixel in the destination rectangle prior to executing either of the BLOCK_PUT functions will be denoted by the variable C. There are sixteen boolean functions of the variables B and C, numbered 0 through 15. The logical operation performed between source and destination rectangles corresponding to each of these values is presented in the ROM Kernel Manual in the chapter entitled 'Graphics Primitives' in the discussion headed 'Copying Rectangular Areas'.

As there is a limited amount of CHIP memory (512K), it is sometimes useful to deallocate a BLOCK previously saved with a BLOCK_GET or to transfer the contents of the BLOCK from CHIP memory to another kind of memory. The BLOCK_CLOSE command allows the deallocation of a BLOCK previously created by a BLOCK_GET command. This operation frees up memory for a later BLOCK_GET. The syntax is:

BLOCK_CLOSE #<block number>

The user program may wish to save the contents of a created BLOCK in another portion of memory or even to save the contents to a disk file for later use. This operation is effected by the BLOCK_SAVE command, whose syntax is:

<error code>=BLOCK_SAVE #<block number> (<ArrayAddress>)

Here, <block number> specifies the BLOCK to be transferred from CHIP memory to the array. <ArrayAddress> is the starting address of an array into which the contents of the BLOCK will be copied. The value of <ArrayAddress> is normally obtained by applying the F-Basic @ operator to any array name. The only requirement that the designated array must satisfy is that it must be large enough to contain the pixel codes for all pixels within the saved rectangle together with the width, height, and depth of the rectangle. Specifically, if W is the width of the saved rectangle in pixels $(X2-X1+1)$ in absolute

mode, or X2 in relative mode), and H is the height of the rectangle ((Y2-Y1+1) in absolute mode, or Y2 in relative mode), and D is the number of bitplanes present in the source window for the saved BLOCK, then the array must contain B bytes, where:

$$B = ((W+7)/8) * H * D + 12$$

After the BLOCK_SAVE, the first three long words of the array contain respectively the width, height, and depth of the saved rectangle. The remainder of the array is packed with the pixel codes of the bit planes corresponding to the rectangle. If an array of type INTEGER is used, each element contains 4 bytes. For an array of type WORD, each element contains 2 bytes, while for an array of type BYTE, each element contains 1 byte. Thus, a BYTE array must contain at least B elements, a WORD array must contain at least (B+1)/2 elements, and an INTEGER array at least (B+3)/4 elements.

After the execution of a BLOCK_SAVE command, the contents of the corresponding array may be written to disk using the PRINT# command. A program may retrieve saved rectangles from disk by using the INPUT# command and reading the data back into a corresponding array. The image may then be transferred to CHIP memory by using the F-Basic BLOCK_RESTORE command, whose syntax is:

```
<error code>=BLOCK_RESTORE #<block number> (<ArrayAddress>)
```

In most respects, BLOCK_RESTORE performs an operation that is the inverse of BLOCK_SAVE. The one difference is that BLOCK_SAVE does not deallocate the BLOCK given by <block number>, while BLOCK_RESTORE, like BLOCK_GET, creates (allocates) a new BLOCK. An <error code> of zero is returned by BLOCK_RESTORE if there exists insufficient available CHIP memory to create a new BLOCK or if a BLOCK with the same <block number> already exists. If BLOCK_RESTORE is successful in the restore operation, a one is returned.

The following program segment illustrates the creation of a BLOCK using BLOCK_GET, the transfer of the contents of that BLOCK to an array using BLOCK_SAVE, and the creation of a new BLOCK containing the same data using BLOCK_RESTORE.

```
PROGRAM ExampleBlock
INTEGER I,J,K,L,BlockBuffer(126)
I=WINDOW #1 (0,0,640,200,0,0,0,0,-1,-1,31,0,-1)
?
? Fill window with imagery using other F-Basic
? graphics functions.
?
J=BLOCK_GET #1(0,30,30,70,70)
K=BLOCK_SAVE #1 (@BlockBuffer)
L=BLOCK_RESTORE #3 (@BlockBuffer)
```



```

?
?   Now, both BLOCKs 1 & 3 store the rectangle's information
?
END

```

Here, $H=W=(70-30+1)=41$. $D=2$ as the window is opened in the default WorkBench screen, which is 2 bitplanes deep. Therefore, from the formula above, $B=504$ bytes. Since BlockBuffer is an INTEGER array, the number of elements required is $504/4=126$ elements.

BLOCK GET and BLOCK PUT make use of the Blitter chip on the Amiga, which is restricted to CHIP memory. These BLOCK operations operate very quickly due to the Blitter. BLOCK SAVE and BLOCK RESTORE utilize the 68000 processor to transfer image data to and from the BLOCK structure to a specified array. These operations are thus relatively slower than BLOCK PUT and BLOCK GET. BLOCK SAVE and BLOCK RESTORE may be used to save and retrieve image data from disk or in systems with more than 512K of memory to free up CHIP memory for the creation of new BLOCKs.

Section 7-Menus In F-Basic

Pull-down menus have become a popular method of creating a user interface for application programs. In this section, the high level approach to creating and manipulating pull-down menus is presented.

A menu consists of a Menu Title and zero or more Menu Items that appear when the Menu Title is selected by the user from the MenuBar at the top of the screen. F-Basic allows the MenuBar to contain from one to ten Menu Titles, as space allows.

Creating a pull-down menu in F-Basic is accomplished through the following sequence of operations:

1. The program must open a window, as described in Section 2 of this chapter.

2. The text for the Menu Title and the Menu Items associated with that menu should be stored consecutively in an array of type TEXT. The TEXT array should be blank filled initially, with the Menu Title as the first element of the array and the Menu Items text stored in the succeeding array elements.

3. The actual data structures used by the system to create the menu are constructed by a call to the F-Basic MENU statement, whose syntax is:

<integer variable>=MENU #<menu number>(<@TextArray>,<Number Of Items>)

Here, <menu number> is an integer in the range 1 to 10, <@TextArray> is an integer specifying the address of the TEXT array containing the Menu Title and Menu Item text, and <Number Of Items> is an integer specifying the number of Menu Items in this Menu. <Number Of Items> does not include the Menu Title in its count. A convenient way to obtain the address of the TEXT array, as required in <@TextArray>, is to apply the F-Basic address operator, @, to the TEXT array name. The value returned by the MENU statement is stored in <integer variable> and represents the address of the associated menu structure, if it was successfully created, and zero otherwise.

The following program segment opens a window and creates a menu titled "MainMenu" which has three Menu Items associated with it. These Items are titled "Option1", "Option2", and "Option3".

```
PROGRAM MenuDemo
INTEGER I,M1
TEXT*12 STR1(15)
DATA (STR1,"MainMenu","Option1","Option2","Option3")
I=WINDOW #1 (0,0,640,200,0,0,0,0,-1,-1,31,@"Window1",-1)
M1=MENU #1 (@STR1,3)
```

A discussion of some technical matters associated with the Amiga's menu structure is necessary to properly achieve the desired pull-down menu interface. First, many Amiga system routines that deal with text strings require that they be terminated with a zero byte. This is the case for the text strings representing the Menu Title and Menu Items. The F-Basic MENU statement described in step 3 above automatically terminates the text strings with a zero byte. The user must be conscious of this and leave at least one unused character at the end of each text array element for this purpose.

Secondly, the number of Menu Titles that will fit across the top of a screen in the MenuBar is determined by two factors, high or low screen resolution, and the number of pixels (horizontally) allocated for each Menu Title. In a high resolution screen, the MenuBar is 640 pixels wide, while it is only 320 for a low resolution screen. F-Basic assigns a default value of 80 pixels as the Menu Title width. With a high resolution screen, this default value limits the number of visible Menu Titles to $640/80 = 8$, and limits the number of characters per Menu Title to approximately 8. To accommodate longer Menu Titles, one must increase the number of pixels per Menu Title from the default value of 80. Likewise, to increase the number of visible Menu Titles on the MenuBar (up to the maximum of ten per window supported by F-Basic) one must decrease the default value of 80. The compiler directive &MENUWIDTH allows the programmer to change the default value

of the Menu Title width. The syntax is:

`&MENUWIDTH <title width in pixels>`

Here, `<title width in pixels>` is an integer specifying the new value of the number of pixels allocated for the width of each Menu Title. This compiler directive should be given before any MENU statements are executed.

Thus far, the process of opening a window, defining a Menu Title and Menu Items, and creating the data structures necessary for pull-down menus have been presented. The next step is:

4. Cause the system to display the MenuBar at the top of the current output window. This is accomplished by the F-Basic statement `MENU_ON`.

A different MenuBar may be associated with each F-Basic defined window. Only the MenuBar associated with the currently active window is displayed when the right mouse menu button is depressed.

Upon creating and activating a pull-down menu, F-Basic provides several ways to interact with and alter the menu structure. A complete description of the way in which user input via the menu/mouse is processed in F-Basic is provided in the next chapter, entitled 'Event Trapping In F-Basic'. The balance of this section is concerned with the ways in which a pull-down menu may be altered. Each of the following operations, except `MENU_CLOSE`, affects only the MenuBar associated with the current output window.

Having attached the Menu Title(s) to the MenuBar, one may desire to disable an entire menu or selected items within a menu. This is appropriate at any time when the choice of certain items within the menu should be prohibited. The disabling of an entire menu or selected items results in their appearance in a 'ghosted' image and renders them inaccessible to the user. This is accomplished with the F-Basic `MENU_DISABLE` statement, whose syntax is:

`MENU_DISABLE #<menu number>`
or `MENU_DISABLE #<menu number> (<item number>)`

The first of the two forms causes the entire menu (in the current output window) associated with `<menu number>` to be unselectable. If `<menu number>` is zero, all menus on the MenuBar are disabled. The second form disables a specific item, `<item number>`, within a specified menu in the current output window. If `<item number>` is zero, all items associated with `<menu number>` are disabled.

The complementary statements:

MENU_ENABLE #<menu number>
or MENU_ENABLE #<menu number> (<item number>)

may be used to re-enable for selection entire menus or specified items within a menu. When menus are created by the MENU statement discussed above, all menus and associated items are by default enabled. Thus, the MENU_ENABLE statement is only appropriate after a corresponding MENU_DISABLE has been executed.

The Amiga Intuition interface which controls the placement of screens, windows, and menus, among other things, also allows Menu Items to be 'checkmarked' for further clarity in alerting the user as to which of the menu selections are currently or have been previously chosen. F-Basic provides the MENU_CHECK statement to allow a programmer access to this convenient tool. The syntax is:

MENU_CHECK #<menu number> (<item number>)

Here, <menu number> is again an integer from 1 to 10 representing a specific menu choice in the current output window. The parameter <item number>, if positive, represents the Menu Item associated with that menu which should be rendered with a leading checkmark. If <item number> is negative, e.g. -3, then this indicates that the item given by the absolute value of <item number>, e.g. 3, should be rendered without the leading checkmark. This last usage is particularly useful when the programmer wants to remove a checkmark from a previously checked Menu Item. The programmer should be aware that the leading checkmark uses the first two character locations of the text string in rendering the checkmark, and thus, it is appropriate not to utilize these spaces for the text of the Menu Item if the checkmark will be later displayed.

The F-Basic statement MENU_OFF removes the user defined menu(s) from the MenuBar of the current output window but retains all menu structures intact. Thus, a later execution of the MENU_ON statement will restore the MenuBar to its previous state without the need for recreating the Menu Title(s) and Menu Item(s).

Menus, like many other features supported by the Amiga, require dynamic memory allocation for their use. In the case of pull-down menus, this memory allocation was accomplished by the F-Basic MENU statement discussed above. When a user program no longer has a need for the menus, the allocated memory should be freed by the use of the F-Basic MENU_CLOSE statement. Unlike MENU_OFF, once a MENU_CLOSE statement has been executed, all information associated with previously defined MENU statements for all windows is lost. The MENU_CLOSE statement should be included at the end of any program which utilizes pull-down menus to free up system resources for later use.

The following F-Basic program provides an example of all of the pull-down menu features discussed in this section.

```

PROGRAM MenuDemo
INTEGER I,M1,M2,M3,M4
TEXT*12 STR1(15),STR2(15),STR3(15),STR4(15)
DATA (STR1,"MainMenu1","Option1","Option2","Option3")
DATA (STR2,"MainMenu2","OPTION 1","OPTION 2","OPTION 3")
DATA (STR3,"MainMenu3","Opt1","Opt2")
DATA (STR4,"MainMenu4","option1","option2","option3","option4")

&MENUWIDTH 100 ; ? Enlarge MenuBar Pixel Width To 100

I=WINDOW #1 (0,0,640,200,0,0,0,0,-1,-1,31,@"Window1",-1)
M1=MENU #1 (@STR1,3)
M3=MENU #3 (@STR3,2)
M2=MENU #2 (@STR2,3)
M4=MENU #4 (@STR4,4)
MENU ON
DELAY (50)

MENU_DISABLE #1 (2) ; ? Disable Item 2 Of Menu 1
DELAY (50)
MENU_ENABLE #1 (2) ; ? ReEnable It
DELAY (50)

MENU_DISABLE #1 ; ? Disable Menu 1
DELAY (50)
MENU_ENABLE #1 ; ? ReEnable It
DELAY (50)

MENU_DISABLE #1 (0) ; ? Disable All Items Of Menu 1
DELAY (50)
MENU_ENABLE #1 (0) ; ? ReEnable All Of Them
DELAY (50)

MENU_DISABLE #0 ; ? Disable All Menus On MenuBar
DELAY (50)
MENU_ENABLE #0 ; ? ReEnable All Of Them
DELAY (50)

MENU_OFF ; ? Remove Menus From MenuBar But Preserve
DELAY (50) ; ? Definitions From Above
MENU_ON ; ? Redisplay MenuBar As Originally Defined
DELAY (50)

STR1(3)=" Option2"&CHAR(0) ; ? CheckMark Items 2&3 Of Menu#1
STR1(4)=" Option3"&CHAR(0) ; ? And Item 1 Of Menu#2...Insert
STR2(2)=" OPTION 1"&CHAR(0) ; ? Two Leading Spaces For Check
MENU_CHECK #1 (2)

```



```

MENU_CHECK #1 (3)
MENU_CHECK #2 (1)
DELAY(50)
MENU_CHECK #1 (-2) ; ? Remove Checkmark From Same
MENU_CHECK #1 (-3) ; ? Three Items
MENU_CHECK #2 (-1)
DELAY(50)

WINDOW_CLOSE #1 ; ? Clean Up
MENU_CLOSE
END

```


Event Trapping In F-Basic

Introduction

The Amiga provides a variety of features that facilitate the development of user interfaces. These include the use of the mouse, menus, and the keyboard. Since the time at which such interactions occur is uncertain, it is useful to have a method that will alert the program when the user wishes to employ any of these mechanisms. F-Basic provides a general control structure, ON...EVENT/END EVENT, to process such interactions.

Section 1-Events In F-Basic

The term 'event' will be used to refer to any of the following:

1. Issuing A Window Close Request,
2. Performing A Menu Selection,
3. Clicking The Left Mouse Button Once,
4. Clicking The Left Mouse Button Twice,
- and 5. Striking A Key On The Console.

An F-Basic program can be made to respond to any one of the above events by including the ON...EVENT/END EVENT structure, whose syntax is as follows:

```
ON <EventName> EVENT
    {Event Processing Instructions}
END EVENT
```

In this syntax, <EventName> may be any of the keywords WINDOW_CLOSE, MENU_SELECT, SINGLE_CLICK, DOUBLE_CLICK, and INKEY. The EVENT structure block {Event Processing Instructions} refers to any sequence of F-Basic instructions appropriate for processing the specified event.

The ON...EVENT/END EVENT structure may be thought of as similar to an IF/THEN/ENDIF block in some respects. The sequence of instructions within the EVENT block are executed conditionally upon the occurrence of the event specified by <EventName>. The EVENT block structure differs from all other control structures in F-Basic in that the execution of the structure is not predicated upon encountering the

structure in the normal course of instruction execution. All other statements in F-Basic are executed only if control is explicitly transferred to that statement by the execution of some branching control structure or, more frequently, if the statement before it has just finished execution. By contrast, the ON...EVENT/END EVENT structure will be executed anytime the specified event occurs provided that control at this time exists within any section of the main program in which 'event checking' has been enabled.

The precise rules governing event checking will be discussed in detail below, but the general idea is the following. With the appropriate organization of an F-Basic main program, the F-Basic system will respond to any of the above events without seriously limiting the area of program control in advance. This means that the main program may be performing useful computations and not merely executing some 'wait' loop when user interaction is pending. For instance, if the ON...EVENT/END EVENT structure is placed at the top of the main program, then event checking will be enabled for all of the succeeding instructions in the main program. Anytime, then, that the specified event occurs while program control is anywhere in the remainder of the main program, control will immediately pass to the specified EVENT block. The instructions in this block are sequentially executed until the END EVENT instruction is encountered, at which time control returns to the place where execution left off at the time of the event.

In this previous respect, the ON...EVENT/END EVENT structure is like the use of a GOSUB. Recall that a GOSUB transfers control to a labelled statement where execution continues until an LRETURN or RETURN statement is encountered, at which time control passes to the first instruction after the GOSUB statement. The difference between an EVENT block and a GOSUB is that no explicit high level statement (like GOSUB <label>) causes the transfer of control to the EVENT block. Rather, control may potentially be transferred to the EVENT block before the execution of any main program statement for which event checking is enabled if the event is detected by the F-Basic system.

For those readers familiar with interrupt processing, the F-Basic ON...EVENT/END EVENT block may be compared to a high level interrupt handler, while event checking may be compared to polling to see if an interrupt has occurred.

Section 2-Activating Events And Enabling Event Checking

An ON...EVENT/END EVENT structure must be encountered at least once during the normal course of execution for event processing to proceed. When the EVENT block is encountered, the corresponding event

is said to be active. Activating the event alerts the F-Basic system to look for the specified occurrence. For example, until an ON MENU_SELECT EVENT block has been encountered, all user attempts at menu selection will be ignored. The following rules govern the placement and construction of EVENT blocks, and a compile time error will result if any of these are violated:

1. ON...EVENT/END EVENT blocks must be encountered in the main program. Additionally, each ON...EVENT statement must eventually be followed by a terminating END EVENT statement.

2. EVENT blocks may not be nested. That is, one EVENT block may not contain another.

3. The branching control structures GOTO and GOSUB may not be used within an EVENT block, as transfer of control out of an ON...EVENT/END EVENT block without first encountering the END EVENT statement is prohibited. All other control structures, including IF, WHILE, REPEAT, and FOR, as well as SUBROUTINE or FUNCTION invocations are permitted.

4. Event checking is automatically disabled while executing the statements comprising an EVENT block. This prevents the system from becoming confused by a new event while processing an old event. When a new event occurs in this fashion, it is queued, and upon the completion of the processing of the current event, it will in turn receive processing by its own ON...EVENT/END EVENT block provided the event is currently an active one.

5. The main program must open at least one custom window, as explained in the previous chapter, in order to receive event reports from the operating system. The F-Basic system was not designed to receive event reports from the default AmigaDOS CLI window.

The term event checking refers to the process of determining whether an activated event has occurred. Event checking is said to be enabled for an F-Basic instruction if the F-Basic system performs an event check before executing that instruction. Event checking is enabled for an instruction if either of the following conditions is satisfied:

1. The instruction occurs within the main program and occurs after (in the sense of the compilation of the source code) the occurrence of an END_EVENT instruction.

2. The instruction occurs after an &EVENT_CHECK 1 compiler directive has been encountered. This directive explicitly enables event checking. In view of the above considerations, the &EVENT_CHECK directive is illegal inside an ON...EVENT/END EVENT structure.

Likewise, the occurrence of an &EVENT_CHECK 0 directive in the source

code will disable event checking until a line containing END EVENT or another &EVENT_CHECK 1 directive is encountered.

As mentioned earlier, encountering an ON...EVENT/END EVENT structure automatically enables event checking for subsequent main program instructions. The capacity to selectively disable and re-enable event checking in the main program (event checking in subprograms is always disabled) is provided in view of the following two considerations. First, event checking involves a certain amount of overhead, and thus, slows program execution. Secondly, certain computations may require execution without the possibility of interruption. For instance, in order to maintain the consistency of data in some database, the process of updating a file may involve data that could be modified by the execution of an EVENT block. In this case, disabling event checking would be appropriate until the file update is complete.

Section 3-More About F-Basic Events And Their Associated System Variables

In the following discussion, the reader is assumed to have some familiarity with the subjects of custom windows and custom menus in F-Basic as presented in the previous chapter.

The five events supported by F-Basic may be divided into two groups. The first set of events, WINDOW_CLOSE, SINGLE_CLICK, DOUBLE_CLICK, and MENU_SELECT are effected by the use of the mouse. The remaining event, INKEY, refers to the action of striking a key on the keyboard. This section will deal in turn with each of these five events.

F-Basic provides a selection of system variables which allow the EVENT block to access critical information pertaining to the event that occurred. The use of these system variables is limited to expressions occurring within the body of an ON...EVENT/END EVENT block. The system variables WINDOW_NUMBER, KEY_NUMBER, MENU_NUMBER, ITEM_NUMBER, MOUSE_X1, MOUSE_Y1, MOUSE_X2, and MOUSE_Y2 may thus be referenced within any EVENT block of the appropriate type. With the exception of WINDOW_NUMBER, the remaining system variables are only meaningful when used within the appropriate EVENT block. Each of these latter system variables will be discussed below in connection with their associated event. WINDOW_NUMBER, however, may be referenced within any EVENT block and will return the F-Basic window number of the window in which the associated event occurred. This is important when more than one custom window is in use.

The WINDOW_CLOSE Event:

The ON WINDOW CLOSE EVENT will be executed whenever the user clicks on the window close gadget of a custom window. The F-Basic number of the window that the user is attempting to close may be obtained from the system variable WINDOW_NUMBER. There may be certain actions that the program may wish to take, such as deallocating system resources, before issuing the F-Basic WINDOW_CLOSE # statement in the EVENT block to actually close the window. Thus, the action of closing a custom window may be initiated by the user but is always accomplished by the programmer through the WINDOW_CLOSE # statement, as the following code segment illustrates:

```
ON WINDOW_CLOSE EVENT
  WHEN WINDOW_NUMBER IS
    [1] <Do Some Stuff>
        WINDOW_CLOSE #1
    [5] <Do Some Other Stuff>
        WINDOW_CLOSE #5
  ENDCASES
END EVENT
```

The SINGLE_CLICK And DOUBLE_CLICK Events

The ON SINGLE_CLICK EVENT and ON DOUBLE_CLICK EVENT structures are triggered by the use of the left mouse button. These events allow the F-Basic program to be notified when the user clicks the left mouse button inside any custom window. The distinction between successive SINGLE_CLICKs and a DOUBLE_CLICK is established by the double click time which is set in Preferences. If the elapsed time between two successive clicks is less than the Preferences double click setting, the event is reported as one DOUBLE_CLICK. Otherwise, the SINGLE_CLICK event structure, if active, will be executed twice.

In addition to the usual system variable, WINDOW_NUMBER, four mouse click system variables are provided for use in these EVENT structures. MOUSE_X1 and MOUSE_Y1 return the window coordinates (relative to the upper left corner of the window) of the mouse upon depressing the left mouse button, while MOUSE_X2 and MOUSE_Y2 return the corresponding coordinates upon releasing the mouse button. These coordinates may be used to identify the initial and final positions in a "mouse dragging" movement. Here, the left mouse button is depressed, the mouse is moved to a new position within the window, and then the left mouse button is released.

The following example illustrates the ease with which an application may allow the user to draw lines in a custom window.

```
ON SINGLE_CLICK EVENT
  IF DrawEnabled THEN
    WHEN WINDOW_NUMBER IS
      [3] WINDOW_OUTPUT #3
      [5] WINDOW_OUTPUT #5
```



```

        ENDCASES
        COLOR_LINE (0,MOUSE_X1,MOUSE_Y1,MOUSE_X2,MOUSE_Y2)
    ENDIF
END EVENT

```

This code assumes that only windows numbered 3 and 5 have been opened, and that a program variable, DrawEnabled, has been set to a boolean true or false (1 or 0) to signal whether drawing is permitted. If DrawEnabled is non-zero, any SINGLE_CLICK event in windows 3 or 5 will produce a line which begins at the point where the left mouse button was depressed and ends at the point at which the left mouse button was released.

The MENU_SELECT Event:

The creation of custom menus in F-Basic was discussed in the previous chapter. After opening a custom window and attaching a menu strip to it, the user may cause Intuition to display that menu by depressing the right mouse button, and may select a menu item by positioning the mouse over the appropriate item and then releasing the button. The ON MENU_SELECT EVENT structure is executed when such a menu choice is made by the user. The system variable WINDOW_NUMBER is defined in the usual way. The system variables MENU_NUMBER and ITEM_NUMBER, when used in the body of an ON MENU_SELECT EVENT block, return the number of the selected menu title and the menu item within that menu title, respectively.

A possible scheme for processing a MENU_SELECT event is illustrated by the following code segment:

```

    ON MENU_SELECT EVENT
        WHEN MENU_NUMBER IS
            [1] WHEN ITEM_NUMBER IS
                [1] <Process Item 1 of Menu 1>
                [2] <Process Item 2 of Menu 1>
                .
            ENDCASES
            [2] <Process Menu 2 Items>
            .
        ENDCASES
    END EVENT

```

This code assumes that only one custom window was available for menu selection as no check of WINDOW_NUMBER was made. Recall that F-Basic supports up to 20 custom windows, each of which may have its own uniquely defined menu bar.

The F-Basic system ignores right mouse button clicks that are not associated with the choice of a specific menu title and item.

The INKEY Event:

The INKEY event is used to respond to keyboard input on a key by key basis. That is, when this event is active, the EVENT block will be executed once for each single keystroke. One may contrast INKEY with the more conventional way of inputting data in F-Basic, the INPUT statement. The INPUT statement will only receive data when it is encountered in the normal course of program execution, whereas INKEY will respond to a keystroke while the system is executing any instruction for which event checking is enabled. When the INPUT statement is executed the system waits until data is entered, and thus, no useful computations can be performed during this time span. This makes INPUT of limited usefulness for 'polling' the keyboard when the time of data entry is unknown. This is, of course, exactly the purpose of INKEY. Finally, INKEY provides a more primitive character by character mechanism for receiving data, while the INPUT statement will read strings of characters into a single variable.

Within an ON INKEY EVENT structure, the ASCII value of the key struck may be referenced by the use of the system variable KEY_NUMBER. The following program segment illustrates how one may report to the user the character value of the key struck and also terminate the execution of the program in the event that the Break (or Control-C) key is struck:

```
PROGRAM EventDemo
INTEGER I,J
I=WINDOW #18(0,0,640,200,0,0,0,0,-1,-1,31,@"Window 18",-1)
.
ON INKEY EVENT
  J=KEY_NUMBER
  IF J=3 THEN STOP
  PRINT "Keystroke Was ",CHAR(J)," In Window # ",WINDOW_NUMBER
END EVENT
.
<Balance Of Main Program>
.
END
```

With regard to the above program, note the following. First, a custom window, WINDOW #18, was opened in the default Workbench screen to receive keyboard strikes. Second, the EVENT block is encountered at least once in the course of normal execution; thus, activating the INKEY event and enabling event checking for all lines following the END EVENT. The value 3 corresponds to the ASCII value of the Break key. Finally, the system variables WINDOW_NUMBER and KEY_NUMBER return the window number (in this case 18 since only one window has been opened) and the ASCII code for the key struck. When a Break key is struck, execution is terminated by the STOP statement.

Caution: The INKEY event and INPUT statements are incompatible in the following sense. Whenever the current output window is a custom window and the INKEY event is activated, any attempt to execute an INPUT statement will confuse the system.

When an F-Basic program utilizes multiple windows, the following somewhat technical considerations about the INKEY event must be taken into account. By design, the keyboard may at any one time be associated with only one custom window. In F-Basic, this is the current output window. As discussed in the last chapter, the current output window is the last window created by a WINDOW # statement, or the window specified by the most recently executed F-Basic WINDOW OUTPUT # statement. Understanding the current output window is straightforward. Recall next that the active window in the display is either the last window created by a WINDOW # statement, or the window specified by the most recently executed F-Basic WINDOW ACTIVE # statement. Also recall that Intuition allows the user to modify the active window in the display (without the program's knowledge) by simply clicking in one of the inactive, or ghosted, windows. In order for the INKEY event to function properly, the active window number and the current output window must coincide. If this is not true, user keystrokes are ignored by the F-Basic system.

RECORD Structures In F-Basic

Introduction

Array names provide one example of referencing a collection of data by using a single variable name. In this chapter, a second mechanism (generally called a record or structure) for referencing a collection of data items via a single variable name is presented. In addition, a mechanism for referencing data items by their memory address, rather than their variable name, is discussed. This mechanism is generally referred to as a pointer.

Records differ from arrays in two respects. First, an array name represents a collection of data items of the same type, while an F-Basic RECORD variable may contain a collection of data objects of differing types. Second, to reference an individual item in an array, one adjoins to the array name a subscript or index which indicates the number of the item in the array list to be referenced. In a RECORD structure, each individual data object within the record is assigned a unique variable name called the field name. To reference such an object, one adjoins the record variable name to the field name. The exact syntax for the record structures supported in F-Basic is discussed in the sections which follow.

Section 1-Creating A RECORD Type

All variables declared in sample programs in previous chapters of this manual have been required to be of one of a limited number of types supplied by the F-Basic system (e.g. INTEGER, REAL, TEXT, etc.). These types will be referred to as system types. By contrast, RECORD variables represent a data type defined by the user; thus, they will be referred to as user defined types. The process of declaring a RECORD variable consists of two steps:

- STEP 1 - Specify the contents of a user defined type.
- STEP 2 - Declare one or more variables to be of that user defined type.

The first step is the more involved process and is the subject of this section.

The syntax for establishing a user defined type is:


```

TYPE <Typename> IS RECORD
    <list of declaration statements>
ENDTYPE

```

One difference between this structure and previous declaration statements is that the latter requires a single program line while a TYPE...IS RECORD statement requires at least three program lines to be fully defined. The <list of declaration statements> stands for one or more declaration statements of the form:

```

<Typename> <list of variables separated by commas>

```

where <Typename> may be either a system type such as INTEGER, REAL, or TEXT, or it may be a user defined type which has been defined in a TYPE...IS RECORD structure previously. Also, <list of variables separated by commas> is a list of scalar variable names, or singly subscripted array declarations. A multiple subscripted array declaration within a TYPE...IS RECORD structure is not permitted.

It is important to remember that this structure is defining a new type and not declaring a variable to be of this type. Thus, a TYPE...IS RECORD structure allocates no storage and is only necessary to define the various types of data that it will store. The first line of this structure, TYPE <Typename> IS RECORD, establishes <Typename> as the name of a new user defined type. Likewise, the variable names that occur in the declarations that make up the body of the TYPE...IS RECORD structure do not represent declared variables, but rather the field names used to select a specified component of the RECORD structure.

Consider the following example:

```

TYPE Complex IS RECORD
    REAL X,Y
ENDTYPE
TYPE Quad IS RECORD
    Complex Z
    INTEGER A(5)
    TEXT*10 QuadName
ENDTYPE

```

These statements establish 2 user defined types, Complex and Quad. Complex is a RECORD consisting of two components whose field names are respectively X and Y. Type Quad is a RECORD having 3 components whose field names are respectively Z, A, and QuadName. Again, note that these statements have not declared any variables to be used in the program, nor allocated any memory storage, but instead have established 2 new data types.

Section 2-Declaring And Referencing RECORD Variables

Once a user defined type has been established using the TYPE...IS RECORD structure, one or more declaration statements may be used to create variables of the desired type. For example, consider the following program segment:

```
PROGRAM RecordExample
TYPE Complex IS RECORD
    REAL X,Y
ENDTYPE
TYPE Quad IS RECORD
    Complex Z
    INTEGER A(5)
    TEXT*10 QuadName
ENDTYPE
Quad Q,P
Complex R,S(3)
INTEGER I,J,K
REAL A,B,C,X,Y
```

Here, the declarations not contained within the TYPE...IS RECORD structure declare program variables and associated types and memory locations for each variable name, while declarations within the TYPE...IS RECORD structure define the field names that are associated with that type. Note also that field names and variable names do not have to be distinct, as A, X, and Y appear as both in the example.

Consider the line 'Quad Q,P' above. This establishes Q and P as program variables of the user defined type Quad. One may think of Q and P as each being a collection of 3 data objects. The first is itself a user defined data type, Complex. The second is an integer array or list of 5 integers, while the third is a text string of length 10.

In order to reference a component or field of such a variable, one employs the following syntax:

<Variable Name>.<Field Name>

Here, <Variable Name> represents a program variable declared as having some user defined type, while <Field Name> represents a field name declared within the body of the corresponding TYPE...IS RECORD structure. For example, Q.QuadName references the 10 character string which is the third component of the variable Q of type Quad. On the other hand, the expressions Quad.QuadName, or Q.X are illegal as Quad is not a variable name and X is not a field name of the type Quad. If

the field name is itself a user defined type, as in the first field of the type Quad, one references the subfields of this field by appending another period and the appropriate subfield name. Thus Q.Z represents a data item of type Complex, while Q.Z.X is of type REAL as X is a field name of type REAL for any item of type Complex. This process of 'nesting' record references inside record references may be continued to any appropriate depth provided these syntax rules are obeyed.

Thus, Q, Q.Z, and Q.Z.X are all valid names of data objects. Q is of type Quad, Q.Z is of type Complex, and Q.Z.X is of type REAL. Any structure such as this, consisting of a variable name of a user defined type with or without field name references, will be henceforth called a record variable reference.

Section 3-Using RECORD References In Assignments And Expressions

One may equate two record references, provided they are of the same type. For example:

```
Q=P
Q.Z=P.Z
Q.QuadName=P.QuadName
Q.A(3)=P.A(1)
Q.Z.X=P.Z.Y
Q.Z.Y=12.89
R=S(2)
```

are all legal record reference equates, while

```
Q=R
Q.Z=R.X
Q.Z=5.1
```

are illegal as the types on either side of the equal sign are not the same.

Although variables of any type may occur in assignment statements, expressions written in F-Basic may only contain record references which result in one of the system types such as INTEGER, REAL, or TEXT. For example, if the above declarations are followed by this program line:

```
Q.Z.X=3.2*P.Z.Y+(1.5-(Q.Z.Y+1.0)^3.1)
```

the expressions and assignment are valid as all record references are of type REAL.

Section 4-Pointers In F-Basic

A pointer is a variable whose value is the address of some other data object. A pointer may contain the address of any data item except one of type PATTERN. Pointer variables are declared by the following syntax:

PTR_TO <TypeName> <list of variable names separated by commas>

Here, <TypeName> may be a user defined type or one of INTEGER, REAL, or TEXT. Pointer declarations, when used within the TYPE...IS RECORD declaration structure, specify that a field name is a pointer type. For example:

```
TYPE LinkedList IS RECORD
  TEXT*10 NodeNAME
  INTEGER NodeID
  PTR_TO LinkedList P
ENDTYPE
LinkedList Q,R
PTR_TO LinkedList PP
PTR_TO INTEGER Pint
INTEGER I,J,K
```

illustrates the use of the pointer declaration both within a TYPE...IS RECORD declaration as a field name (PTR_TO LinkedList P) and as a declaration statement which creates a program variable of type pointer (PTR_TO LinkedList PP).

A pointer reference consists of a variable name of type pointer, a variable name of type pointer followed by one or more field references, or a record reference with at least one field name of type pointer. For example, PP is a variable name of type pointer, PP.NodeNAME is a field reference using a pointer, and Q.P or Q.P.NodeNAME are record references containing a field name of type pointer. If a pointer reference terminates in a variable or field name of type pointer, like PP or Q.P, then the pointer reference refers to the value of the pointer. In other words, it refers to the address of the data object to which it points. On the other hand, if the pointer variable or a field name of type pointer is followed by a field reference, like PP.NodeNAME or Q.P.NodeNAME, then the pointer reference refers to the value of the data object to which it points.

The syntax of pointer references in F-Basic, as described above, is similar to that in the language ADA, but is different from that

found in C and Pascal. These require respectively the operators * and ^ to be used in conjunction with the pointer variable name in order to reference the value of the data object instead of the address of the data object. The F-Basic system tells the difference between these desired outcomes simply by determining whether the variable name of type pointer is followed by a field reference. The following example illustrates the differences in pointer references between F-Basic and these other languages:

F-Basic Line:	PP.NodeNAME="a"
Ada Line:	PP.NodeNAME="a"
Pascal Line:	PP^.NodeNAME='a'
C Line:	(*PP).NodeNAME='a'
	or PP->NodeNAME='a'

Section 5-Assigning Values To Pointers And Dynamic Memory Management

The simplest way of assigning a value to a pointer involves the use of the @ operator. Recall that @<variable name> returns the memory address of <variable name>. Thus, using the declarations above, one may write:

```
PP=@Q
PP=@R
PInt=@I
PInt=@Q.NodeID
```

causing the value of PP to be the address of the variable Q or R respectively, and the value of PInt to be the address of the variable I or the NodeID field of the variable Q, respectively.

A second and more sophisticated way of assigning values to pointer variables involves the creation of data objects without the use of an associated variable name. These will be referred to as unnamed or dynamic data objects.

The F-Basic system function NEW_REC, and system subroutine DISPOSE_REC operate in conjunction with pointers to create and destroy data objects with which no variable name is associated. A dynamic or unnamed data object must thus be referenced through the use of pointers. The syntax for creating such a data object is:

<pointer to user defined type>=NEW_REC (<user defined type name>)

For example:

PP=NEW_REC(LinkedList)

creates a dynamic or unnamed data object of type LinkedList. The structure of this object is, of course, identical to that of the named data objects Q and R of type LinkedList. The value of PP is the address of this newly created data object and the components of that object may be referenced as PP.NodeNAME, PP.NodeID, and PP.P, just as the components of Q are referenced as Q.NodeNAME, Q.NodeID, and Q.P.

F-Basic allows the destruction of data objects, thus freeing the memory which they occupy, by the use of the system subroutine DISPOSE_REC. The syntax is:

DISPOSE_REC (<pointer to user defined record type>)

For instance, to destroy the previously created unnamed data object pointed to by PP, the statement:

DISPOSE_REC(PP)

should be used.

The syntax employed in dynamic memory management in F-Basic differs from that in C and Pascal. Programmers familiar with these languages should equate C's MALLOC and Pascal's NEW functions to F-Basic's NEW_REC, while associating C's FREE and Pascal's DISPOSE procedures to F-Basic's DISPOSE_REC system subroutine.

Care must be taken not to overwrite the value of a pointer using NEW_REC or a pointer assignment until the associated data object has been removed using DISPOSE_REC. Otherwise, the memory allocated by NEW_REC for this data object will be frozen and unusable by any application until the system is re-booted. For example:

```
{BadNews}
  PP=NEW_REC(LinkedList)
  .
  .
  PP=NIL
```

will have exactly this effect if the lines executed between these two statements do not involve a DISPOSE_REC(PP) call. Notice that after allocating space for the unnamed data object from the operating system's free memory pool, the address of that memory (the value of PP) is changed. Thus, the location of the unnamed data object created in the first statement is thought by the system to still be in use, but the programmer has lost its location in memory by changing the value of PP. This memory is now effectively frozen, and will not be allocated to any other application.

Section 6-DATA Statements Involving RECORD Variables

DATA statements may be used in the main program to initialize the fields of any program variable of a user defined type. The syntax is identical to that used to initialize an array with a DATA statement, that is:

DATA (<record reference>,<list of values separated by commas>)

Like initializing arrays, if the list of values is too long to fit on a single line, one may continue the list of values on a new line using the syntax:

DATA (<list of values separated by commas>)

This construction may be repeated until the list of values required for the initialization of the RECORD structure has been exhausted.

There are two restrictions on <record reference> imposed by the constraint that DATA statements are processed by the FastCom compiler and do not generate any run-time code. First, <record reference> may not directly or indirectly contain a subfield of type pointer. Secondly, <record reference> may not contain directly or indirectly any subfield representing an array. As is usual with F-Basic programs, this constraint also means that DATA statements which initialize RECORD variables must appear within the <declaration part> of the main program. A type is said to occur directly in a record reference if it is declared explicitly in the defining TYPE...IS RECORD structure. A type is said to occur indirectly in a record reference if it may be obtained by repeated application of the field operation (.<field name>).

The <list of values separated by commas> represents a list of literals or F-Basic CONSTANTS of type INTEGER, REAL, or TEXT. Consider the following example:

```
PROGRAM RecordExample
TYPE Complex IS RECORD
    REAL X,Y
ENDTYPE
TYPE Quad IS RECORD
    Complex Z
    INTEGER A
    TEXT*10 QuadName
ENDTYPE
Quad Q,P
```



```

Complex R,S(3)
DATA (Q,2.5,1.73,15,"Example")
DATA (P.Z.X,2.551)
DATA (P.Z.Y,1.773)
INTEGER I,J,K
REAL A,B,C,X,Y

```

Note especially the use of the DATA statement in initializing the program variable Q of user defined type Quad. Even though the first field of any variable of type Quad is itself a user defined variable (type Complex here), the first entry in the list of values in this DATA statement is of type REAL. When using DATA statements to initialize program variables whose RECORD structure contains fields that are of a user defined type, F-Basic treats these variables as if any field of user defined type is replaced by its corresponding definition. This process is repeated until fields of a system type are obtained. Thus, from the viewpoint of a DATA statement, a variable of type Quad appears as the following 'virtual' type:

```

TYPE Quad IS RECORD
  REAL X,Y
  INTEGER A
  TEXT*10 QuadName
ENDTYPE

```

This is the reason that the entries in the DATA statement above contain 2 real, 1 integer, and 1 string value separated by commas.

Section 7-More On RECORDS And Pointers

The library macros INC and DEC, discussed in the chapter entitled 'F-Basic Library Functions and Macros', when applied to any variable of type pointer, have particularly useful results. INC will add (DEC will subtract) the size of the corresponding data objects in bytes to (from) the current value of the pointer. Consider the following code segment:

```

TYPE Complex IS RECORD
  REAL X,Y
ENDTYPE
Complex A(5)
PTR TO Complex P
P=@A ; ? P initialized to first element of A
INC(P) ; ? The value of P is increased by 16

```

Note that the INC statement leaves the pointer P at the address of

A(2), as P was incremented by 16. Pointers declared as type PTR_TO INTEGER, PTR_TO REAL, or PTR_TO TEXT can be used with INC and DEC in this manner to expedite operations involving arrays of data of these corresponding types.

Program variables of user defined types, when passed as parameters to F-Basic subprograms, are passed by address rather than by value. For a full comparison of these mechanisms, refer to the chapter entitled 'Subprograms in F-Basic', Section 7. Remember that pass by address implies that any modification to the parameter in the subprogram also modifies the corresponding value in the calling program.

Recall that F-Basic subprograms are either of type SUBROUTINE or FUNCTION. F-Basic supports FUNCTION subprograms of both user defined and pointer type. For example:

```
PROGRAM RecordExample
TYPE Complex IS RECORD
    REAL X,Y
ENDTYPE
LOCAL
    Complex Q
SUBPROGRAM
    Complex F
Q=F(2)
PRINT Q.X
END
FUNCTION F
PARAMETER
    INTEGER N
F.X=FLOAT(N)
F.Y=FLOAT(N+1)
END
```

In this example, it is acceptable to use a call to a RECORD valued function in a RECORD equate, as in 'Q=F(2)'. Likewise, it is permissible to reference the FUNCTION name followed by a field reference in the subprogram, as in 'F.X=FLOAT(N)' and 'F.Y=FLOAT(N+1)'. However, F-Basic does not support a subfield reference of a call to a RECORD valued function. For example:

```
PRINT F(2).X
```

is illegal and would result in an unpredictable and undesirable outcome.

F-Basic Library Functions And Macros

Introduction

When using the many system library functions and macros supported by F-Basic, it is important to insure that the number and type of the parameters passed to these functions is correct. The FastCom compiler checks both the number and type of parameters passed to these routines to verify that the usage is correct. Failure to exactly follow the requirements outlined below will result in a compile time error.

An alphabetized list of all the library functions and macros supported by the F-Basic system is provided in Appendix 4. In this chapter, the functions and macros are grouped according to data type and usage. One of them, EXCHANGE, works on all data types but is discussed in Section 3.

Section 1-Operations On Integers

The following library functions produce integer results:

`<result>=BCHG(<bit number>,<value>)`

The argument `<bit number>` is an integer expression ranging from 0 to 31 which specifies the bit number to be complemented in the integer expression given by `<value>`. The result of complementing the specified bit is returned by the function BCHG.

`<result>=BCLR(<bit number>,<value>)`

The argument `<bit number>` is an integer expression ranging from 0 to 31 which specifies the bit number to be cleared (set to zero) in the integer expression given by `<value>`. The result of clearing the specified bit is returned by the function BCLR.

`<result>=BSET(<bit number>,<value>)`

The argument `<bit number>` is an integer expression ranging from 0

to 31 which specifies the bit number to be set (set to one) in the integer expression given by <value>. The result of setting the specified bit is returned by the function BSET.

<result>=BTST(<bit number>,<value>)

The argument <bit number> is an integer expression ranging from 0 to 31 which specifies the bit number to be tested in the integer expression given by <value>. The value of the specified bit (zero or one) is returned by the function BTST.

DEC(<integer variable>)

DEC(<integer variable>,<integer expression>)

DEC is a subroutine and function which decrements a variable name representing an integer scalar or a singly subscripted integer array reference. <integer expression>, when present, determines the value to be subtracted from <integer variable>. If <integer expression> is not present, DEC subtracts one from <integer variable>. DEC is equivalent to, but faster than, <integer variable>=<integer variable>-<integer expression>.

DEC, when applied to a variable of type PTR_TO INTEGER, PTR_TO REAL, or PTR_TO <RECORD type> decrements the pointer by the length of the data object to which it points. For more details, see the chapter entitled 'RECORD Structures In F-Basic'.

DELAY(<delay count>)

DELAY is a subroutine which causes the processor to pause for a length of time determined by <delay count>. The integer expression <delay count> specifies the time to delay in approximately 1/10 of a second intervals. Thus, the line DELAY(50) will cause a delay of approximately 5 seconds.

<result>=EVEN(<integer expression>)

The function EVEN returns 1 if <integer expression> is even (exactly divisible by 2) and 0 otherwise.

<result>=FALSE

FALSE is a system constant which, when used in any logical or arithmetic expression, returns the value 0 (logical false).

`<result>=HIWORD(<integer expression>)`

The function HIWORD returns an unsigned integer obtained from the upper 16 bits (bits 16 through 31) of `<integer expression>`.

`<result>=IABS(<integer expression>)`

`<result>=ABS(<integer expression>)`

Either of the functions IABS or ABS return an integer representing the absolute value of `<integer expression>`.

`INC(<integer variable>)`

`INC(<integer variable>,<integer expression>)`

INC is a subroutine and function which increments a variable name representing an integer scalar or a singly subscripted integer array reference. `<integer expression>`, when present, determines the value to be added to `<integer variable>`. If `<integer expression>` is not present, INC adds one to `<integer variable>`. INC is equivalent to, but faster than, `<integer variable>=<integer variable>+<integer expression>`.

INC, when applied to a variable of type PTR_TO INTEGER, PTR_TO REAL, or PTR_TO <RECORD type> increments the pointer by the length of the data object to which it points. For more details, see the chapter entitled 'RECORD Structures In F-Basic'.

`<result>=INTSORT(<array name>,<first element>,<last element>)`

INTSORT is a function which sorts an integer (32 bit only) array beginning with element `<first element>`, and terminating with `<last element>`. Suppose that LIST is an integer array of 100 elements. Then:

`K=INTSORT(LIST,20,70)`

will sort elements 20 through 70 of LIST in ascending order. The value returned by the function is 1 if the sort was successful. The value returned is zero if `<first element>` is greater than `<last element>`, or if `<first element>` or `<last element>` is outside of the declared bounds of the array.

`<result>=LOWORD(<integer expression>)`

LOWORD returns an unsigned integer obtained from the lower 16 bits (bits 0 through 15) of <integer expression>.

<result>=MAXINT

MAXINT is a system constant whose value is the largest positive integer $((2^{31})-1)$ representable in F-Basic.

<result>=NIL

NIL is a system constant which returns the value zero when used in any expression.

<result>=ODD(<integer expression>)

The function ODD returns a value of 1 if the value of <integer expression> is odd, and 0 otherwise.

<result>=RANDOM()

The function RANDOM returns a randomly generated positive 32 bit integer. Thus, the values returned by RANDOM range from 1 to MAXINT. RANDOM is especially fast, and uses a generation algorithm that will only repeat after MAXINT successive calls.

RANDOMIZE(<integer expression>)

RANDOMIZE is a subroutine which, when referenced, uses the value of <integer expression> in combination with the current date and time to re-initialize the random integer generator RANDOM, and the random real generator RANDOMREAL. This method of 'seeding' the random number generator prevents the same sequence of numbers from being generated on successive runs of a program which uses random numbers.

During debugging, it is sometimes desirable to repeatedly generate the same sequence of 'random' numbers. If <integer expression> is negative, the F-Basic system does not combine this value with the current date and time. The effect will be that the same sequence of numbers are returned by RANDOM or RANDOMREAL calls during successive executions.

<result>=SWAP(<integer expression>)

SWAP returns an integer value obtained by exchanging the high and low word of <integer expression>. Thus, bits 0-15 of <integer expression> are exchanged with bits 16-31.

<result>=TRUE

TRUE is a system constant which, when used in any logical or arithmetic expression, returns the value 1 (logical true).

Section 2-Operations On Real Numbers

All routines and macros in this section accept real parameters and return real values.

<result>=ABS(<real expression>)
<result>=RABS(<real expression>)

The result returned by these functions is the absolute value of <real expression>.

<result>=FRAC(<real expression>)

FRAC returns the decimal fraction part of <real expression>. For example, FRAC(3.1415) has the value .1415 .

<result>=ACOS(<real expression>)
<result>=ASIN(<real expression>)
<result>=ATAN(<real expression>)

These three functions return the inverse cosine, inverse sine, and inverse tangent, respectively, of <real expression>. The value returned is measured in radians.

<result>=COS(<angle>)
<result>=SIN(<angle>)
<result>=TAN(<angle>)
<result>=COT(<angle>)
<result>=SEC(<angle>)
<result>=CSC(<angle>)

These six functions return the cosine, sine, tangent, cotangent, secant, and cosecant, respectively, of the real expression <angle>. The parameter <angle> is interpreted in radian measure. Apart from 68000 systems with floating point co-processors, the above transcendental functions are believed to be the fastest available in any comparable system.

```
<result>=EXP(<real expression>)  
<result>=LN(<real expression>)
```

These functions return, respectively, the natural exponent and natural logarithm of <real expression>. As with the trigonometric functions, these functions have been optimized for high performance.

```
<result>=COSH(<real expression>)  
<result>=SINH(<real expression>)  
<result>=TANH(<real expression>)
```

These functions return, respectively, the hyperbolic cosine, hyperbolic sine, and hyperbolic tangent of <real expression>.

```
<result>=MAXREAL  
<result>=MINREAL
```

MAXREAL and MINREAL are system constants returning, respectively, the largest and smallest positive floating point number representable in F-Basic. The values of these two numbers are so extreme that they are not printable. For more details on their size, refer to the chapter entitled 'Variables And Data In F-Basic'.

```
<result>=PI
```

PI is a system constant which, when used in any arithmetic expression, returns the value 3.14159265.

```
<result>=RANDOMREAL()
```

The function RANDOMREAL returns a number strictly greater than zero and strictly less than one. RANDOMREAL is a very fast library function for generating random real numbers which cycles only after MAXINT calls. For details on seeding the random number generator, see RANDOMIZE in the previous section.

<result>=SQR(<real expression>)
<result>=SQRT(<real expression>)

These functions compute, respectively, the square and the square root of <real expression>. In the case of SQRT, the parameter must be strictly non-negative. SQR(x) is equivalent to, but faster than, x*x.

Section 3-Operations Involving Text Expressions

EXCHANGE (<data1>,<data2>)

EXCHANGE is a system subroutine that swaps the values of the two variables <data1> and <data2>. It is equivalent to, but generally faster than, the following three instructions:

```
HOLD=<data1>
<data1>=<data2>
<data2>=HOLD
```

Here, <data1> and <data2> represent two scalars, array references, or RECORDs of the same type and size. All data types except PATTERN may be EXCHANGED.

<result>=FILLCHAR(<string to fill>,<repeat string>)

The function FILLCHAR fills <string to fill> with as many copies of <repeat string> as will fit. For example, if STR is a TEXT string of declared length 12, the expression FILLCHAR(STR,"ABC") will cause the value of STR to be "ABCABCABCABC". The value returned is the integer 1 upon successful termination of FILLCHAR.

<result>=LENGTH(<string expression>)

The function LENGTH returns an integer specifying the position of the last non-blank character in <string expression>. The value returned by LENGTH is always less than or equal to the declared length of <string expression>.

<result>=LEVEL(<target string>,<search string>,<length of substring>)

Here, <target string> is a concatenation of one or more substrings of length <length of substring>. <search string> is a TEXT

string which is to be searched for any occurrence of one of the substrings making up <target string>. The value returned by LEVEL is the integer position of the first character of the right-most substring of <target string> encountered in <search string> at level zero with respect to parentheses (i.e. not contained within parentheses). For example:

```
K=LEVEL("ANDNOTTOP","THIS AND THAT IS NOT THE TOP",3)
```

will return the value 26, while:

```
K=LEVEL("ANDNOTTOP","THIS AND THAT IS (NOT THE TOP)",3)
```

will return the value 6. This function is useful in parsing expressions containing parentheses.

```
<result>=LOCASE(<text string>)  
<result>=UPCASE(<text string>)
```

These functions accept a text string containing both upper and lower case letters. They modify their parameter by transforming all upper case letters to lower case (or lower case to upper case). The value returned, <result>, is an integer representing the number of characters modified.

```
<result>=REPLACE(<string1>,<string2>,<string3>)
```

This routine replaces every occurrence of <string1> in <string3> by <string2>. If <string1> is shorter than <string2>, <string3> will contain trailing blanks. If <string1> is longer than <string2>, any characters in the resulting <string3> past its declared length are discarded. The value returned by the REPLACE function has no significance.

```
<result>=SPACE(<integer number of spaces>)
```

This function returns a text string containing a sequence of <integer number of spaces> blanks, where <integer number of spaces> is any integer in the range 1 to 120. For example, PRINT "ABC", SPACE(10),"DEF" will cause the output of "ABC" to be separated from "DEF" by ten spaces.

```
<result>=SUBSTRING(<string1>,<string2>)
```

If <string1> is a substring of <string2> then the function

SUBSTRING returns the position in <string2> of the first character of <string1>. SUBSTRING returns zero if <string1> is not a substring of <string2>. For example:

```
K=SUBSTRING("IS A","THIS IS A TEST")
```

returns a value of 6.

```
<result>=TXTSORT(<array name>,<first element>,<last element>)
```

TXTSORT is a function which sorts a text array beginning with element <first element>, and terminating with element <last element>. Suppose that LIST is a TEXT array of 100 elements. Then:

```
K=TXTSORT(LIST,20,70)
```

will sort elements 20 through 70 of LIST in ascending order. The value returned by the function is 1 if the sort was successful. The value returned is zero if <first element> is greater than <last element>, or if <first element> or <last element> is outside of the declared bounds of the array.

Section 4-Functions That Convert One Data Type To Another

This section contains library functions which convert between types INTEGER, REAL, and TEXT.

```
<result>=ASCII(<character>)
```

The value of ASCII is an integer equal to the ASCII code for the single character string <character>. For example, ASCII("A") returns an integer value of 65. A complete ASCII table is presented in the Appendix to this manual. ASCII and CHAR are inverse functions.

```
<result>=BASE(<text string>,<integer base>)
```

Here, <text string> represents a number expressed in the base <integer base>. The value of the function BASE is an integer representing the decimal equivalent of <text string>. For example:

```
K=BASE("A2",16)
```


will return an integer value of 162. <integer base> may represent any base from 2 to 37. For a more detailed discussion of the representation of numbers to bases other than 10, refer to the chapter entitled 'Variables And Data In F-Basic'.

<result>=CHAR(<integer expression>)

CHAR returns a text string of length one representing the character whose ASCII code is represented by the integer <integer expression>. The value of <integer expression> must be in the range 0 to 127. A complete ASCII table is presented in the Appendix to this manual. CHAR and ASCII are inverse functions.

<result>=DIGSTRING(<integer expression>)

The value of DIGSTRING is a TEXT string representing the decimal integer <integer expression>. For example, DIGSTRING(123) returns the three character string "123". DIGSTRING and IVAL are inverse functions.

<result>=FLOAT(<integer expression>)

The function FLOAT converts <integer expression> into an equivalent real number.

<result>=HEX(<integer expression>)

The function HEX converts <integer expression> into a TEXT string representing the hex (base 16) equivalent of the integer.

<result>=INT(<real expression>)

The function INT converts <real expression> into an equivalent integer by discarding the fractional part of <real expression>. For example, INT(3.6415) returns an integer value of 3.

<result>=IVAL(<text string>)

IVAL returns a decimal integer equivalent to the integer represented by <text string>. For example, IVAL("123") returns an integer value of 123. IVAL and DIGSTRING are inverse functions.

<result>=RVAL(<text string>)

RVAL returns a real number representing the floating point equivalent of the parameter <text string> whose characters represent a decimal number. For example, RVAL("123.456") returns a real number whose value is 123.456. If the parameter <text string> is a text variable, rather than a text literal, the text variable must contain at least one trailing blank.

Section 5-Functions That Interact With AmigaDOS

The following functions allow a user program to directly interact with the Amiga's operating system to obtain desirable results.

<result>=ALLOCATE(<number of bytes needed>)

<result>=DEALLOCATE(<number of bytes to release>,<pointer to block>)

ALLOCATE requests a block of free memory whose size is specified by <number of bytes needed>. If sufficient memory exists, the function is successful and returns the address of the beginning of the memory block. Otherwise, <result> is set equal to zero.

A call to DEALLOCATE frees a previously allocated block of memory. The first parameter specifies the number of bytes in the block, and the second parameter specifies the address of the beginning of the block. The address of the block is returned by the previous ALLOCATE function call. The operating system does no error checking on a DEALLOCATE memory call, and thus an error in either parameter will cause unpredictable and undesirable results.

<result>=ALLOCATE_WITH(<number of bytes needed>,<memory type>)

ALLOCATE_WITH performs the same function as ALLOCATE, except one may request memory with specified attributes. The parameter <memory type> is an integer with the following possible values:

<u><memory type></u>	<u><requested memory attributes></u>
1'16	Public Memory
2'16	Chip Memory
4'16	Fast Memory
10000'16	Memory Cleared (Filled With 0's)
20000'16	Allocates Memory With The Largest Possible Address

One may request any combination of these attributes by forming the sum of the appropriate integers. For instance, <memory type> equal 10000'16 + 4'16 will request both Fast Memory filled with 0's. The value returned by ALLOCATE_WITH has the same meaning as that in ALLOCATE.

<result>=AVAILMEM(<memory type>)

AVAILMEM returns the number of bytes of available memory of the type specified by <memory type>. The acceptable values of the parameter <memory type> are as specified in the description of the ALLOCATE_WITH function above.

<result>=OPENLIB(<library name>,<minimum version number>)
<result>=CLOSELIB(<pointer to library>)

A call to the function OPENLIB returns <pointer to library> and opens the specified library provided that the resident version of the library is greater than or equal to the integer <minimum version number>. The parameter <library name> is a string specifying one of the legal AmigaDOS libraries, as referenced by the ROM Kernel Manual. A <result> of zero indicates that either the library or a sufficient version is not available.

The act of opening a library usually means that the library is read into memory from the floppy disk. Thus, when one no longer needs the library, its resources should be freed for the use of others. This is accomplished through the CLOSELIB function. Here, the parameter <pointer to library> is the value returned by the corresponding OPENLIB call. The <result> of a CLOSELIB call is always zero. For example:

```
INTEGER ExecBase,Temp
ExecBase=OPENLIB("exec.library",1)
    <executable statements>
Temp=CLOSELIB(ExecBase)
```

opens the Exec library and places the pointer to it in the variable ExecBase. Upon completion of its use, the program closes the Exec library to free resources.

<result>=COMSTRING(<text variable>)

The function COMSTRING must be invoked as the first line of the <executable part> of an F-Basic program. A call to COMSTRING will fill

the string represented by <text variable> with the "command tail" of the last DOS command issued. The <result> returned is the integer 1 if the command tail was present and transferred to the <text variable>, and 0 if the command tail was not utilized or left blank. For example, if the program containing the call to COMSTRING was stored in a file named 'test.bin', and was invoked by the DOS command:

```
1> test.bin This is the command tail...
```

then <text variable> would contain the string "This is the command tail..." after the call to COMSTRING was executed. If the number of characters in the command tail exceeds the declared length of <text variable>, then the command tail is truncated to fit. If the number of characters in the command tail is less than the declared length of <text variable>, then the remainder of this string is blank filled.

This function is extremely useful in user applications where it is desired to provide optional forms for invoking the application.

```
<result>=DOSEXECUTE(<DOS command string>)
```

A call to DOSEXECUTE allows a user program to issue any valid DOS command. <DOS command string> is a string containing the desired AmigaDOS command. The function returns a value of 1 if the command is a legal DOS command and is successfully executed and 0 otherwise. For example:

```
I=DOSEXECUTE("cd df1:")
```

will issue the command to change the current directory to the external floppy disk drive, and store the result in the integer variable I. A complete discussion of all legal DOS commands is available in 'The AmigaDOS User's Manual'.

```
<result>=DOSTIME(<pointer to DATESTAMP block>)
```

A call to DOSTIME writes the current date and time into a block of 12 bytes whose address is specified by <pointer to DATESTAMP block>. The 12 byte block is divided into three 4-byte integers. The first integer contains the number of days elapsed since a unique date fixed by the AmigaDOS operating system. The second integer contains the number of minutes that have elapsed in the current day. The third and final integer contains the number of ticks elapsed in the current minute (1 tick equals 1/50 seconds). For example, using F-Basic's RECORD structures:

```
PROGRAM TIMEIT
```



```

INTEGER I
TYPE DateStamp IS RECORD
    INTEGER DAYS,MINUTES,TICKS
ENDTYPE
DateStamp Start,TEnd
SUBPROGRAM
    SUBROUTINE ElapsedSeconds
        .
        .
        I=DOSTIME(@Start)    ;    ? get starting time
        .
        .                    ;    ? stuff to time here
        .
        I=DOSTIME(@TEnd)     ;    ? get ending time
    
```

sets Start.DAYS to the elapsed days, Start.MINUTES to the elapsed minutes, and Start.TICKS to the elapsed ticks, all at the 'starting' time. Conversely, the fields of TEnd are filled with the elapsed values at the 'ending' time. The value returned by DOSTIME, I in this example, is always 1.

When combined with an appropriate subroutine, referred to as ElapsedSeconds above, these two event times can be manipulated to yield the elapsed time in seconds between the two calls to DOSTIME. In the above example, a call to ElapsedSeconds(@Start,@TEnd) would print the elapsed seconds between start and stop time, provided ElapsedSeconds was defined as:

```

SUBROUTINE ElapsedSeconds
PARAMETER
    PTR_TO DateStamp A,B
LOCAL
    INTEGER TotSeconds
    TotSeconds=(B.DAYS-A.DAYS)*4320000+(B.MINUTES
        -A.MINUTES)*3000+(B.TICKS-A.TICKS)
    TotSeconds=TotSeconds/50.0
    PRINT "Elapsed Time Was",TotSeconds," Seconds"
    GRETURN
END

```

<result>=FILELENGTH(<filename>)

The function FILELENGTH returns the length of the file (in bytes) specified by the text string <filename>. As discussed in the chapter entitled 'File Input And Output In F-Basic', the string <filename> may contain an alternate AmigaDOS path to examine files on other than the current directory. A return value of zero indicates that the file was not found or is empty.

Accessing Amiga ROM Libraries From F-Basic

Introduction

The Amiga supports a variety of system libraries. These libraries permit the user to directly access various system services such as memory allocation, management of windows, sound, graphics, etc. F-Basic provides an easy and convenient way to utilize any of the approximately 450 public Amiga system library routines.

The Amiga system routines which accept pointers to character strings require that the character string be followed in memory by a zero byte. If the user program employs literal strings, e.g. "exec.library", then the compiler automatically places a zero byte at the end of the text string. This represents the more common case. If, however, the user program passes a pointer to a text variable, the F-Basic interface to the Amiga system routines will locate the last non-blank character in the text string and place a zero byte in the following character position. It is thus necessary in this circumstance, such as with the F-Basic functions OPENLIB and NARRATE, to insure that there is at least one trailing blank in the text string.

When directly accessing ROM Kernel routines which require pointers to text strings as parameters (using the &SYSLIB compiler directive discussed below), the user program must explicitly include a zero byte terminating the text string if the string is not a text literal.

Section 1-A Summary Of Available Amiga Libraries

Appendix A of the ROM Kernel Manual contains an exhaustive listing of Amiga library routines. The appendix specifies the name of each library routine, the library that the routine belongs to, the parameter list for the routine, and a brief description of the function of the routine.

Section 2- F-Basic Access To An Amiga Library Routine

The process of invoking an Amiga system library routine in F-Basic consists of the following four steps:

1. The user program must contain the compiler directive:

&SYSLIB 1

This alerts the FastCom compiler that references to system libraries follow. The statement need only be placed once in the program. Including this directive slightly slows the speed of compilation. The directive &SYSLIB 0 can be used to alert the FastCom compiler that no further system library references follow.

2. For each library referenced, an F-Basic open library call must be made. The purpose of this call is to instruct the operating system to load the corresponding Amiga library from disk into RAM memory and return a pointer to that library's memory location. Some libraries are already in memory, and in this case the pointer to the library's existing location is returned. The syntax of the open library call is:

<pointer to library>=OPENLIB(<library name>,<minimum version #>)

Here, <library name> is a string containing the name of the desired library. <minimum version #> is an integer specifying the minimum acceptable version number of the library. A <minimum version #> of 0 will indicate that any version is acceptable. The returned value <pointer to library> will be zero if no acceptable version of the desired library is available on the current directory.

3. After the user has familiarized himself with the parameters of the desired library routine, the user program may then invoke the specified library routine in the following manner for a function:

<int.var>=<routine name>(<pointer to library>,<parameter1>,<parameter2>..)

or for a subroutine:

<routine name>(<pointer to library>,<parameter1>,<parameter2>..)

Here, <pointer to library> is the value obtained in step 2. <parameter1>,<parameter2>,... represents the parameter list of the desired library function as presented in Appendix A of the ROM Kernel Manual.

NOTE: All F-Basic calls to Amiga system routines contain

one more parameter, <pointer to library>, than the corresponding documentation suggests for the library routine. This extra parameter is always the first listed. Thus, even a system function or subroutine with no parameters will be invoked from an F-Basic program with the single parameter <pointer to library>.

4. After the user program has finished accessing routines from a given library, the library should be closed using F-Basic's close library routine. The syntax is:

`<result>=CLOSELIB(<pointer to library>)`

Here, <pointer to library> is again the value returned by the open library call in step 2. Failure to close a library may result in the loss of the effective memory available to your program. Closing a library when it is no longer needed frees the memory and resources allocated to it for the use of others in the Amiga's multi-tasking environment. The returned value <result> is always zero.

Care must be taken in typing the names of system libraries and their routines. Upper and lower case usage is significant, and the spelling of such names must exactly match the form presented in the ROM Kernel Manual. For long routine names, only the first twenty (20) characters should be used. An alternative source of these library routine names is contained on the F-Basic disk in the file FastSysLib. The contents of this file should not be modified, as it is used by the Fastcom compiler to properly encode calls to system routines.

Section 3-An Example Program

The following sample program illustrates the four step process outlined above. AvailMem is a function belonging to the executive library "exec.library". Its function is to return the amount of available memory belonging to a category specified by a single parameter (as shown in Appendix A of the ROM Kernel Manual). Here, the parameter chosen is 1 to indicate 'public' memory. Note that the F-Basic version of the call to AvailMem contains two parameters, a pointer to the executive library and the parameter expected by the system library routine itself.

```
PROGRAM AvailableMemory
INTEGER ExecPtr,J
&SYSLIB 1
```



```
ExecPtr=OPENLIB("exec.library",0)
IF NOT ExecPtr THEN
    PRINT "Error Opening Exec Library"
    STOP
ENDIF
J=AvailMem(ExecPtr,1)
PRINT J
J=CLOSELIB(ExecPtr)
END
```


Interacting Directly With The 68000 Processor

Section 1-Accessing Processor Registers

F-Basic allows direct access to the 8 data registers (D0 through D7), the 8 address registers (A0 through A7), and the program counter register (PC). The syntax for a register reference is:

:<register symbol>

where <register symbol> is one of D0 through D7, A0 through A7, or PC. When used in an expression :<register symbol> returns an integer representing the contents of the corresponding register. For example:

PRINT :D6+(:D7),:A7*5+2

When used on the left side of an assignment statement, the value of an integer expression may be stored in the corresponding processor register. For example:

:D5=3000 LSL 3

will load the 68000 data register D5 with the 32 bit 2's-complement representation of the integer 3000 left shifted 3 places. Assignments to the PC register are not permitted.

Operations involving the registers of the 68000 processor must be used WITH GREAT CAUTION. The FastCom compiler uses registers D0 through D5 in evaluating arithmetic expressions. Registers D6 and D7 contain important information for any statement that occurs within a FOR Loop, and their value must not be altered. The address registers A0, A2, A4, and A7 contain important system pointers, and therefore, must be preserved without modification. Registers A1, A3, A5, and A6 are frequently used by the FastCom compiler to store selected integer and real variable values in order to improve program execution speed. Thus, in general, there are no free processor registers within the context of an F-Basic program. Modifying these registers without first protecting their contents is sure to provide undesirable and unexpected results, including frequent trips from the Guru.

However, safe use of processor registers may be achieved by any of the following methods:

1. In a module not employing real arithmetic, subprogram, or library calls, registers D2 and D3 are free.

2. In any program segment not contained within a FOR loop, registers D6 and D7 are free.
3. If the statement UNOPTIMIZE occurs as the first statement of the <declaration part> of any program module, registers A1, A3, A5, and A6 are free. It should be noted that the UNOPTIMIZE statement has other adverse effects, which are fully discussed in the chapter entitled 'Suggestions For More Effective Use Of The F-Basic System'.
4. If the F-Basic statements SAVE_REG, SAVE_A_REG, RESTORE_REG or RESTORE_A_REG are used to protect critical processor register values before an application program modifies their contents. The proper use of these statements is the subject of the next section.

Section 2-Saving And Restoring Processor Register Contents

The statements SAVE_A_REG and SAVE_REG respectively store registers D6/D7/A0-A6 or D0-D7/A0-A6 on the system stack. The related commands RESTORE_A_REG and RESTORE_REG respectively restore these values from the system stack to the registers being protected. One may bracket any sequence of instructions that modify processor registers between the appropriate SAVE and RESTORE commands with the result being the preservation of the necessary values of the specified registers. Although the FastCom compiler will generate no errors on the unbalanced use of SAVE and RESTORE commands, such use will result in unpredictable and undesirable program behavior. Every SAVE_A_REG statement must be balanced by a closing RESTORE_A_REG command, and every SAVE_REG statement must be balanced by a closing RESTORE_REG command.

Although SAVE and RESTORE protect the contents of registers, any modifications to A0 or A4 within the scope of a SAVE/RESTORE pair may impair the validity of F-Basic variable references. For example:

```
SAVE_A_REG
:A0=1000
:A4=2000
Sum=Sum+1
RESTORE_A_REG
```

will cause the reference to the program variable Sum to perform improperly. This is because A0 and A4 contain the important system pointers used by F-Basic to locate variables.

Section 3-Inserting Machine Code Within F-Basic Programs

At any point in the <executable part> of an F-Basic program module it is permissible to directly insert machine code using the INSERT statement. The syntax is:

INSERT "<even number of hex digits>"

Here, <even number of hex digits> represents the hex equivalent of one or more 68000 machine code instructions. For example:

INSERT "4DFA00064840"

would insert the 68000 instructions 'LEA 6(PC),A6' and 'SWAP D0' into the executable code produced by the compiler for the current program at the point where the INSERT statement was encountered.

The compiler does not check to make sure that the text literal string contains only hex digits (0..9 A..F). Inclusion of extraneous characters will produce unpredictable and undesirable results.

Section 4-Executing Direct Machine Code

The command EXEC <integer expression> calls a machine code program whose absolute address is specified by <integer expression>. This is accomplished by generating a 68000 machine code JSR instruction whose destination is <integer expression>. To utilize this instruction, one may perform the following four steps:

1. Allocate a block of memory with either the F-Basic ALLOCATE or ALLOCATE_WITH function.
2. Read a file containing 68000 machine code into the allocated block of memory using the indirect reference operators (#,\$, or ^).
3. Execute the code using the above EXEC command.
4. Deallocate the memory block using DEALLOCATE.

Suggestions For More Efficient Use Of The F-Basic System

Introduction

This chapter contains a few suggestions for improving the execution speed of F-Basic programs.

Section 1-Optimized Versus Unoptimized Variable Storage

Unless otherwise directed, the F-Basic compiler assigns the first four declared LOCAL INTEGER variables or the first two declared LOCAL REAL variables in each program module to 68000 processor registers, rather than to memory locations. These are referred to as optimized FAST variables. Referencing an optimized variable is four times faster than referencing the same variable when stored in a memory location. Thus, the use of optimized storage variables speeds the evaluation of arithmetic expressions. To take maximum advantage of this built-in feature, one should follow two rules:

1. INTEGER variables that are referenced most frequently, particularly those variables occurring within looping structures, should be declared before other INTEGER variables.
2. If a module contains more than four INTEGER variables, and contains FOR/NEXT loops, one should not declare the FOR loop indices among the first four optimized variables. This is because FOR loop indices are always kept in 68000 processor registers distinct from those used for the storage of optimized variables, and are thus always already optimized by the F-Basic system.

There is a downside to the use of optimized FAST variables. When an F-Basic program module calls another F-Basic program module, the optimized variables within the 68000 processor must be saved on the system stack before the call and restored from the system stack after returning from the subprogram. This adds to the overhead of subprogram calls and thus, slows their execution when a module employs optimized variables.

In evaluating the trade-off between faster variable references versus slower subprogram calls, there are two considerations. First, in any module which has no calls to other subprograms, optimized

variables are always advantageous. Secondly, if a module calls other subprograms, the break even point occurs when there are two optimized variable references for every subprogram call. If there are more subprogram calls than pairs of optimized register references, optimized storage degrades program performance. As this latter condition may not be easily determined by inspection of the source program, the F-Basic system provides a simple command for disabling optimized variable storage, UNOPTIMIZE. The UNOPTIMIZE command must be the first statement issued after the PROGRAM, SUBROUTINE, or FUNCTION header. In any module in which the advantage of the use of optimized storage is in doubt, one can simply compare the execution time of the program with and without the UNOPTIMIZE command.

The UNOPTIMIZE command is also useful in modules which directly access 68000 processor registers, as discussed in the previous chapter. By issuing this command in any program module, the 68000 registers A1, A3, A5, and A6 are free for programmer access and use for the duration of the module.

When using indirect reference operators (#, \$, and ^), the speed of indirect references is greatly improved when applied to optimized variables. For examples of this technique, see the sieve programs contained in the timing test file on the sample program disk included with your F-Basic package.

Section 2-Arrays And Pointers

As noted above, the efficiency of code contained within looping structures is especially important. One operation that occurs frequently within loops is an array reference. It is more efficient to replace array references with pointer references, particularly if the pointer is an optimized FAST variable. Consider the following two examples:

```
PROGRAM First
INTEGER I,J,K,A(10000)
FOR I=1 TO 100
  FOR J=1 TO 10000
    A(J)=J
  NEXT J,I
END
```

```
PROGRAM Second
INTEGER I,J,K,A(10000)
FOR I=1 TO 100
  K=@A
```



```

FOR J=1 TO 10000
    #K+=J
NEXT J,I
END

```

These illustrate the replacement of an array reference by a pointer reference. The outer loop in the above programs has been included to render the execution time measurable. In the inner loop, the array reference A(J) has been replaced by a pointer reference #K+=J. This results approximately in a two-fold increase in execution speed. For the meaning of the syntax #K+=J, refer to Section 4 of this chapter.

Section 3-Speeding Up Integer Multiplies And Divides

The &QUICK 1 compiler directive, discussed in the chapter entitled 'Operators, Expressions, & Assignments In F-Basic', will dramatically speed up programs that make intensive use of integer multiply, divide, or mod. The use of this directive is appropriate roughly when the magnitude of the integer operands of these operators is less than 32,767 (16 bit or WORD size integers).

Section 4-Auto Increment And Auto Decrement In Indirect Reference

The indirect reference operators #, \$, and ^, when used on the left hand side of an assignment statement and applied to a variable name, possess an especially useful option. The option will be illustrated and explained in connection with the # operator. The syntax:

```
#<integer variable name>+ = <expression>
```

is equivalent to, but faster than:

```
#<integer variable name>=<expression>
INC(<integer variable name>,4)
```

while,

```
#<integer variable name>- = <expression>
```

is equivalent to, but faster than:

```
DEC(<integer variable name>,4)
```


#<integer variable name>=<expression>

If # is replaced by \$ or ^, the value 4 should be replaced respectively by 2 or 1. As illustrated in the above code segment, these increment/decrement options are especially fast when applied to optimized FAST variables.

Pattern Matching In F-Basic

Introduction

This chapter introduces the powerful string manipulation operations that are provided by F-Basic and are similar to those found in the language SNOBOL4, which was developed at Bell Telephone Laboratories, Inc. in the early 1960's. Fundamentally, 'pattern matching' is the process of examining a subject string for the occurrence of a substring with specific desired characteristics. These characteristics are described in F-Basic with a 'pattern' or template. Although this chapter presents a complete discussion of the basics of the pattern matching facilities supported by the FastCom compiler, the reader may desire a more extensive treatment. Such completeness can be found in the book entitled 'The SNOBOL4 Programming Language', by R.E. Griswold, Prentice-Hall, 1971.

Table 1 compares the operator symbols used in F-Basic with those used in SNOBOL4.

<u>F-Basic Symbol</u>	<u>SNOBOL4 Symbol</u>	<u>Description</u>
*	<space>	Pattern Concatenation
+	!	Pattern Alternation
?	<space>	Pattern Match
.	.	Conditional Assignment
^	\$	Immediate Assignment

Table 1 - Pattern Operator Symbols In F-Basic And SNOBOL

Section 1-Patterns

A pattern is an ordered list of strings of characters. A pattern variable is a symbol which denotes that list. Pattern variables are declared in the declaration part of the program using a PATTERN statement:

```
PROGRAM TestPattern
INTEGER I,J,K,L
TEXT Collie*20
PATTERN PET,PET_NAME,PRONOUN
```


The last statement shown in PROGRAM TestPattern establishes three pattern variables: PET, PET NAME, and PRONOUN. The simplest pattern is an ordered list containing no strings. This pattern is denoted by NULL, and is a 'system' or 'predefined' pattern. The next simplest patterns are lists consisting of a single text string. For example, the statement:

PET="Cat"

defines PET to be the list consisting of the single string "Cat". A text literal such as "Cat", or a text variable such as Collie (above) is called a simple pattern.

More complex patterns can be constructed from system patterns and simple patterns by using the binary operations Pattern Concatenation (denoted in F-Basic by *) and Pattern Alternation (denoted in F-Basic by +). Pattern Alternation combines two separate patterns into a single combined pattern. Essentially, this means that two ordered lists of strings are combined together to form one ordered list of strings. For example:

PRONOUN="My" + "Your"

establishes the pattern PRONOUN as an ordered list of two strings, {"My" or "Your"}. More generally, let P1 represent the list {S1,S2,...,Sk} and P2 represent the list {S1',S2',...,Sm'}. Here P1 represents a list of k strings S1 through Sk, and P2 a list of m strings S1' through Sm'. The pattern alternation operation:

P1+P2

produces a new pattern consisting of the ordered list {S1,S2,...,Sk,S1',S2',...,Sm'}. The significance of the precise ordering assigned by the alternation operator will be explained in the next section. Because of the ordering, the pattern P1+P2 is not equivalent to the pattern P2+P1. We now redefine PET as:

PET="Cat" + "Dog" + "Fish"

The pattern PRONOUN+PET represents the ordered list {"My","Your","Cat","Dog","Fish"}.

The next pattern operation is Pattern Concatenation. With P1 and P2 defined as above, the pattern P1*P2 is an ordered list {S1&S1', S1&S2',...,S1&Sm',S2&S1',S2&S2',...,S2&Sm',S3&S1',S3&S2',...,S3&Sm',..., Sk&S1',Sk&S2',...,Sk&Sm'}. Here the symbol & represents string concatenation in F-Basic as previously defined. As a reminder, "My" & "Cat" & "yours" in F-Basic is equivalent to the character string "MyCatyours". Because S1&S1' is not equivalent to S1'&S1, and due to

ordering, the pattern $P1 * P2$ is not equivalent to $P2 * P1$. Suppose we defined:

```
PET_NAME="Bill" + "Fred"
```

Then the pattern:

```
PRONOUN* " *PET* " *PET_NAME
```

represents the following list of twelve (12) strings: {"My Cat Bill", "My Cat Fred", "My Dog Bill", "My Dog Fred", "My Fish Bill", "My Fish Fred", "Your Cat Bill", "Your Cat Fred", "Your Dog Bill", "Your Dog Fred", "Your Fish Bill", "Your Fish Fred"}. This example gives a hint of the expressive power of pattern matching. In general, if $P1$ is a list of K strings, and $P2$ is a list of L strings, then the pattern $P1 + P2$ is a list of $K + L$ strings and the pattern $P1 * P2$ is a list of $K * L$ strings.

There are many additional predefined or system patterns (like NULL) that are supported by F-Basic. These will be fully discussed in Section 5.

The expression defining a pattern may reference the pattern. Such a pattern is said to be 'recursive'. Consider the following F-Basic statement:

```
PET="AB" * PET
```

The pattern variable PET has been used 'recursively' as it occurs both on the right and left hand side of the above equation. Recursive patterns must be used with great care. The above pattern is a list that consists of a single infinite string {"ABABABABABABABAB..."} . To see this clearly, replace the variable PET on the right hand side of the equation by its definition, obtaining:

```
PET="AB" * "AB" * PET
```

Repeating this substitution yields:

```
PET="AB" * "AB" * "AB" * PET
```

Continuing this substitution leads to an infinite string of "AB"'s. The difficulty with this definition of PET is that it has no way of terminating. A reference during execution to a pattern with this difficulty will produce undesired results, including a possible system crash.

A better recursive definition of PET uses the system pattern NULL:

PET="AB" * (PET + NULL)

The pattern PET now represents a list with an infinite number of strings, each with finite length. Each string in the list consists of a finite number of repetitions of the two character sequence "AB". This list is ordered using the explanation above so that longer sequences of "AB"'s precede shorter ones. Thus, the pattern PET represents the ordered list {..."ABABABAB", "ABABAB", "ABAB", "AB"}. To understand this, we may use the left distributive law (which is valid for patterns) and the fact that "AB"*NULL is simply equal to "AB". Thus, PET may be redefined as:

PET="AB" * PET + "AB"

Substituting again for PET yields:

PET="AB" * "AB" * PET + "AB" * "AB" + "AB"

Note that the far right of this expression now contains a single "AB" sequence, and just to the left of that is the "ABAB" member of the list. Repeating this process indefinitely leads to the alternation of sequences of all lengths of "AB"'s with the longer sequences preceding the shorter sequences. Compare this recursive definition of PET with the previous one. In general, a recursive definition must contain an alternate subpattern which is free of recursion and serves as a terminating condition. In the most recent definition of PET the + NULL serves this purpose. As you will see, recursive patterns can be extremely powerful and useful if employed properly.

Section 2-Pattern Matching

Patterns, defined above as ordered lists of strings, by themselves may seem curious but of little use. The third fundamental operation on patterns is 'Pattern Matching'. The syntax of pattern matching is:

<string expression> ? <pattern>

The left operand represents a string. Text literals, text variables, or any text valued expressions are acceptable. The pattern matching operation invoked with ? returns an integer value. The pattern match consists of a search of the list of strings comprising the pattern until a string in the list is found which is identical to an 'initial' substring of the left operand. The integer value returned by the pattern match is the number of characters in the substring. For example:

"CATCH" ? ("DOG" + "CAT")

returns the value three. The pattern on the right consists of the list {"DOG","CAT"}. The pattern matcher first compares "DOG" with an initial substring (one which begins with the first character) of "CATCH". As "DOG" is not an initial substring of "CATCH", the pattern matcher proceeds to the next string in the list defined by the pattern. It will either eventually find a string in the pattern list which matches an initial substring, in which case the search terminates and the number of characters in the substring is returned, or the list of strings is exhausted. In this latter case, the pattern match operation returns zero.

In the above example, success occurs with the second string in the list as "CAT" coincides with the first three letters of "CATCH". The significance of the order can be seen from the following example:

"CATCHER" ? ("DOG" + "CATCH" + "CAT")

Here, the pattern match succeeds with the second string in the pattern list and returns the value five. However,

"CATCHER" ? ("DOG" + "CAT" + "CATCH")

also succeeds with the second string in the pattern list but returns the value three.

The construction of F-Basic patterns that are appropriate for any given string manipulation problem requires both skill and subtlety, which unfortunately is best learned through the trial and error process common to your first exposure to many other new ideas. The reader interested in pattern matching is encouraged to experiment with the various pattern definitions, structures, and 'system' patterns provided in F-Basic. The experience of the authors of the language is that although there is a somewhat steep learning curve associated with their use, the ultimate ease of manipulating text data provided through pattern matching is unlike any other high level language feature available.

The following program segment provides a simple instance of the three operations on patterns that have been discussed so far:

```
PROGRAM PatternExamp
INTEGER I,J
PATTERN Day,Month,Year,Date
TEXT*30 STRINGVAR
Day=NUMBER
Month="Jan"+"Feb"+"Mar"+"Apr"+"May"+"Jun"+"Jul"
Month=Month+"Aug"+"Sept"+"Oct"+"Nov"+"Dec"
Year="88"
Date=Day * "/" * Month * "/" * Year
```



```

{Prompt}
INPUT "Enter Date in DD/MMM/88 Format" STRINGVAR
I=LENGTH(STRINGVAR)
J=STRINGVAR ? Date
IF I<>J THEN
    PRINT "Error In Date Entry...Please Follow Format"
    GOTO Prompt
ENDIF

```

NUMBER is a system pattern that is fully defined in Section 5. It's definition is such that it will match any string of digits (0..9). The pattern match assigns to J the number of characters in the input string that coincide with the Date pattern. As defined earlier, LENGTH is an F-Basic library function that returns the position of the last non-blank character in a text string. The I<>J error check serves two purposes. First, the format of the input string must correspond to the specified pattern, otherwise J would be set to zero. In other words, the pattern match must succeed. Second, the test detects the presence of any extraneous characters in the input string. These characters would not match the Date pattern and the condition I<>J would once again be true. Some examples of user input and expected results follow:

<u>User Input:</u>	<u>Results:</u>
"06/Jan/88"	I=9 J=9 Input Acceptable
"1923/Jan/88"	I=11 J=11 Input Acceptable
"8/Feb/88 "	I=8 J=8 Input Acceptable
"0*/Jan/88"	I=9 J=0 User must re-enter
"13/Fer/88"	I=9 J=0 User must re-enter
"08/JAN/88"	I=9 J=0 User must re-enter
"15/Dec/88 yr"	I=12 J=9 User must re-enter
"14/Dec /88"	I=10 J=0 User must re-enter

Section 3-Conditional And Immediate Assignment Operators

It is often useful to have more detailed information from the pattern matching process than simply the integer representing the number of characters in the initial substring that match. For instance, in the program segment concluding the previous section, if the date was entered properly it might be useful to extract the month and day from the input string. This may be done using the conditional assignment operator, which is denoted by the period '.' in F-Basic. The syntax of the conditional assignment operation follows:

<pattern>.<text variable>

The operator '.' will store in the text variable the characters of the substring that match the pattern provided that the entire pattern match process succeeds. Immediate assignment, denoted by '^' in F-Basic, is similar to conditional assignment except that the transfer of characters to the text variable occurs even if the entire pattern match process is unsuccessful (returns zero). To illustrate this difference, recall the program segment at the end of Section 2. Below, the program was modified by adding two TEXT variables, DAYSTRING and MonthString, each of which is blank filled with a DATA statement. Additionally, the pattern Date has the subpattern Day conditionally assigned to the text variable DAYSTRING and the subpattern Month has been conditionally assigned to the text variable MonthString.

```
PROGRAM PatternExamp
INTEGER I,J
PATTERN Day,Month,Year,Date
TEXT*30 STRINGVAR,DAYSTRING,MonthString
DATA (DAYSTRING," "), (MonthString," ")
Day=NUMBER
Month="Jan"+"Feb"+"Mar"+"Apr"+"May"+"Jun"+"Jul"
Month=Month+"Aug"+"Sep"+"Oct"+"Nov"+"Dec"
Year="88"
Date=Day.DAYSTRING * "/" * Month.MonthString * "/" * Year
{Prompt}
INPUT "Enter Date in DD/MMM/88 Format" STRINGVAR
I=LENGTH(STRINGVAR)
J=STRINGVAR ? Date
IF I<>J THEN
    PRINT "Error In Date Entry...Please Follow Format"
    GOTO Prompt
ENDIF
.
.
.
```

Suppose that at the prompt, one entered 15/May/88. Then, after the pattern matching operation had finished, I and J would both equal nine, DAYSTRING would equal "15", and MonthString would equal "May". However, if one entered "15/May/A7", then I would equal nine, J would equal zero, and both DAYSTRING and MonthString would be left blank. This happens because although the subpatterns Day and Month succeeded the overall pattern match operation failed (Year did not match "A7") and J was therefore assigned the value zero.

To compare and contrast immediate assignment with conditional assignment, suppose the pattern Date is instead defined as follows:

```
Date=Day^DAYSTRING * "/" * Month^MonthString * "/" * Year
```


Again, given a user input of 15/May/88, I and J would both equal nine, DAYSTRING would equal "15" and MonthString would equal "May". This result using immediate assignment is the same as the result using conditional assignment, because the overall result of the pattern matching operation was success (J=9). Now, if the user input was "15/May/A7", I would be set to nine, and J would be set to zero as the pattern match failed. However, DAYSTRING would be set to "15" and MonthString would be set to "May". The character transfer to DAYSTRING and MonthString occurred even though the overall pattern match failed. This property is the only difference between the immediate and conditional assignment operators.

Section 4-Anchored And Unanchored Pattern Matching

As discussed in Section 2, the syntax of a pattern match is:

`<string expression> ? <pattern>`

Patterns, as defined in Section 1, are ordered lists of strings. The pattern match procedure searches the ordered list of strings until it finds a string that matches an initial substring of the left operand. This process is actually referred to as 'anchored mode pattern matching', as the match is required to begin with the first character of the left operand. If this requirement is waived, then we obtain a process referred to as 'unanchored mode pattern matching'. For example:

`"a123b" ? NUMBER`

will fail in the anchored mode and return the value zero. In the unanchored mode, the pattern match would succeed (matching characters 2 through 4) and return the value three. One further difference between pattern matching in F-Basic and in SNOBOL4 is that the default pattern matching mode in F-Basic is anchored, while in SNOBOL4 the default is unanchored. The compiler directive `&ANCHOR 0` will signal the pattern matching procedure that further pattern matches should be conducted in the unanchored mode. The compiler directive `&ANCHOR 1` returns the pattern matcher to anchored mode on pattern matches encountered thereafter.

Section 5-System Patterns

In this section, the 'system' (pre-defined) patterns provided by F-Basic are enumerated. Each system pattern is accompanied with a brief discussion of its definition and uses.

A. ARB

ARB is a list of strings beginning with NULL followed by all strings of finite length ordered by increasing length. That is, shorter strings precede longer strings. ARB is useful primarily in combination with other patterns. For example:

STRING ? "A" * ARB * "B"

will match any string beginning with "A" and containing a "B". If STRING="AB", then the pattern match operation will return two. If STRING="ACDFB", then the pattern match operation will return five. Finally, if STRING="ADEdb", then ? will return zero.

One should generally avoid the use of more than one ARB in a single pattern. Once the pattern matching operation has reached the second ARB, it will never backtrack to an earlier stage of pattern matching, as ARB always succeeds. This will cause alternative pattern matches for the first occurrence of ARB to be lost. Lastly, no pattern should end with ARB, as ARB always initially matches the NULL string.

B. ABORT

ABORT is a pattern which causes failure and immediate termination of the pattern matching operation. For example:

STRING ? "A" * ABORT + ARB * "B"

will match any string containing a "B" in which the letter "A" is not the first character. If STRING="ABE", then the pattern match will return zero. If STRING="TOBACCO", then the ? operation would succeed and return the value three.

C. NAME

NAME represents an ordered list of all strings that begin with a letter of the alphabet (either upper or lower case) or an underline character '_', and contain only letters, underlines, or digits. The list is ordered by decreasing length-longer strings precede shorter strings. Put another way, NAME will match the longest initial substring satisfying the above condition. For example:

"Jack_2+Jill" ? NAME

will succeed and return a value of six.

D. NULL

As mentioned in Section 1, NULL represents a list containing no strings. It is peculiar in that it always succeeds and yet matches zero characters. For example:

"AB" ? "A" * ("C" + NULL)

will succeed and return the value one.

E. NUMBER

NUMBER represents a list of all strings composed entirely of digits. The list is ordered from longer strings to shorter strings. Thus, NUMBER matches the longest string of digits possible. For example:

"123+5" ? NUMBER

will succeed and return the value three.

F. SUCCEED

SUCCEED always succeeds and matches no characters. This system pattern generates no alternatives.

G. FAIL

The pattern FAIL causes failure of the subpattern to which it is attached, but unlike ABORT does not immediately terminate the pattern matching process.

H. ANY(<string>)

ANY is the first system pattern that has a parameter. Here, <string> is any literal string. ANY matches a single character if the character belongs to the parameter string and otherwise fails. For example:

"AlB" ? "A" * ANY("123") * "B"

will succeed and return the value three. The system pattern NUMBER, defined above, is also equivalent to:

NUMBER=ANY("0123456789") * (NUMBER + NULL)

I. NOTANY(<string>)

NOTANY matches a single character, provided it does not belong to the literal string parameter. Consider a pattern NODIG, defined as:

NODIG=NOTANY("0123456789") * (NODIG + NULL)

NODIG will match the longest substring not containing a digit.

J. SPAN(<string>)

SPAN matches the longest string of characters each of which belongs to the literal string. The system pattern NUMBER could also be defined as:

NUMBER=SPAN("0123456789")

and would again match the longest string of digits possible.

K. BREAK(<string>)

BREAK matches the longest string of characters no one of which belongs to the literal string. The pattern NODIG defined above could be more simply defined by:

NODIG=BREAK("0123456789")

and would again match the longest substring containing no digits.

L. LEN(<positive integer>)

LEN(K) matches the next K characters of the input string. For example, the pattern "a" + LEN(3) * "b" will match any five character string beginning with an "a" and ending with a "b".

M. BAL(<two character literal string>)

BAL is equivalent to an ordered list of all character strings that are 'balanced' with respect to the two character literal string passed as its parameter. BAL is most frequently used to verify that an expression is 'balanced' with respect to structures such as "{}", "[]", "<>", and "()". The following strings are balanced with respect to "()": "a", "a+b", "(a)+b", "(((a))+b)", and "((a+b)+(a+b))". Some examples of strings that are not balanced with respect to "()" include ")a(", "((a+b)", "a+(b))", and "((()))(".

A formal definition of balanced would include several ideas. First, a string which does not contain parentheses is balanced with respect to "()". Second, an expression of the form "(" & <balanced expression> & ")" is balanced with respect to "()". Finally, the concatenation of any two balanced strings with respect to "(" is itself balanced with respect to "()". The ordered list of balanced strings represented by BAL is ordered so that shorter strings precede longer strings. In other words, BAL matches the shortest balanced substring possible. For example:

"A(B+C)D" ? BAL("()")

returns the value one as "A" is the shortest balanced substring available. However,

"{(A+B) + C}" ? BAL("{}")

will return the value eleven.

N. LPOS(<non-negative integer>)

LPOS(K) matches zero characters and will succeed only if the length of the substring matched prior to encountering this pattern contained exactly K characters. For example, the pattern "A" * ARB * LPOS(4) matches all strings of length four that also begin with "A".

O. RPOS(<non-negative integer>)

RPOS(K) matches zero characters and will succeed only if the length of the substring matched prior to encountering this pattern is exactly N-K characters, where N is the total number of characters in the string to be matched. In particular, RPOS(0) will succeed only if the entire subject string matches the pattern. For example:

"123A" ? NUMBER * RPOS(0)

will fail and return zero, while

"123A" ? NUMBER * RPOS(1)

will succeed and return the value three.

P. BETWEEN(<two character literal string>)

BETWEEN will match any single character whose ASCII value is greater than or equal to the ASCII value of the first character of the

literal string parameter, and is less than or equal to the ASCII value of the second character of the literal string parameter. For example, BETWEEN("AZ") will match any uppercase letter of the alphabet and return a value of one, while BETWEEN("az") will match any lower case letter of the alphabet. One could define a pattern called DIGIT to be BETWEEN("09"), and this would match any single digit.

Q. GOSUB(<statement number or statement label>)

GOSUB always succeeds, matches zero characters, and causes a temporary exit of the pattern matching process, in order to execute any sequence of F-Basic statements terminating in a RETURN or LRETURN. Upon encountering one of these statements, program control returns to the pattern matcher, which continues processing the pattern from the point at which it encountered the GOSUB pattern. Some caution must be exercised when using the GOSUB pattern as the pattern matching procedure is not re-entrant. This means that no pattern matching operations should be performed when executing statements connected with the GOSUB. The following program section illustrates the use of GOSUB:

```
PROGRAM GoSubTest
INTEGER Sum
DATA (Sum,0)
PATTERN P,DIGIT
TEXT*120 Str,Str1*1
DIGIT=BETWEEN("09")^Str1 * GOSUB(AddDigit)
P=DIGIT * (P + NULL)
INPUT "Enter String of Digits" Str
IF Str ? P THEN
    PRINT "Sum of Digits Is:",Sum
ELSE
    PRINT "A String of Digits Was Not Entered"
ENDIF
STOP

{AddDigit}
Sum=Sum+ASCII(Str1)-ASCII("0")
LRETURN

END
```

The pattern DIGIT will match any single digit. The digit is immediately assigned to Str1 and then control is transferred to the GOSUB at label AddDigit. The pattern P is a recursive pattern which will match any string of digits. At label AddDigit, Sum is incremented by the integer value of the digit. This process is then repeated for each digit in the input string.

R. FENCE

FENCE matches zero characters and always succeeds. Once FENCE has been encountered, no string in the list represented by the pattern will be considered unless it agrees with the substring matched before encountering FENCE. This explanation hides a significant amount of the details of the pattern matching process from the user, and the reader desiring a more complete discussion is encouraged to read the SNOBOL4 reference mentioned in the introduction to this chapter.

The ASCII Character Set

The following table lists the characters in the ASCII character set and their decimal and hexadecimal values.

<u>ASCII Character</u>	<u>Decimal Value</u>	<u>Hexadecimal Value</u>
nul	0	0
soh	1	1
stx	2	2
etx	3	3
eot	4	4
enq	5	5
ack	6	6
bel	7	7
bs	8	8
ht	9	9
nl	10	A
vt	11	B
np	12	C
cr	13	D
so	14	E
si	15	F
dle	16	10
dc1	17	11
dc2	18	12
dc3	19	13
dc4	20	14
nak	21	15
syn	22	16
etb	23	17
can	24	18
em	25	19
sub	26	1A
esc	27	1B
fs	28	1C
gs	29	1D
rs	30	1E
vs	31	1F
sp	32	20
!	33	21
"	34	22
#	35	23
\$	36	24
%	37	25
&	38	26

' () * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ? @ A B C D E F G H I J K L M N O P Q R S T U V W X Y

39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89

27 28 29 2A 2B 2C 2D 2E 2F 30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58 59

Z	90	5A
[	91	5B
\	92	5C
]	93	5D
^	94	5E
~	95	5F
Tab	96	60
a	97	61
b	98	62
c	99	63
d	100	64
e	101	65
f	102	66
g	103	67
h	104	68
i	105	69
j	106	6A
k	107	6B
l	108	6C
m	109	6D
n	110	6E
o	111	6F
p	112	70
q	113	71
r	114	72
s	115	73
t	116	74
u	117	75
v	118	76
w	119	77
x	120	78
y	121	79
z	122	7A
{	123	7B
	124	7C
}	125	7D
~	126	7E
del	127	7F

The Default Colors Of The 32 Amiga Color Registers

As explained in the chapter entitled 'High Level Graphics In F-Basic', the use of a particular color register without a previous COLOR_DEFINE statement will cause the default color of that register to be displayed. The following table presents the default values for each of the 32 registers.

<u>Register</u>	<u>Red Value</u>	<u>Green Value</u>	<u>Blue Value</u>	<u>Resulting Color</u>
0	0	5	10	Dark Blue
1	15	15	15	White
2	0	0	2	Black
3	15	8	0	Orange
4	0	0	15	Blue
5	15	0	15	Purple
6	0	15	15	Cyan
7	15	15	15	White
8	6	2	0	Dark Brown
9	14	5	0	Red-Orange
10	9	15	1	Lime Green
11	14	11	0	Gold
12	5	5	15	Blue
13	9	2	15	Violet
14	0	15	8	Blue-Green
15	12	12	12	Gray 12
16	0	0	0	Black
17	13	2	2	Red
18	0	0	0	Black
19	15	12	10	Tan
20	4	4	4	Gray 4
21	5	5	5	Gray 5
22	6	6	6	Gray 6
23	7	7	7	Gray 7
24	8	8	8	Gray 8
25	9	9	9	Gray 9
26	10	10	10	Gray 10
27	11	11	11	Gray 11
28	12	12	12	Gray 12
29	13	13	13	Gray 13
30	14	14	14	Gray 14
31	15	15	15	Gray 15

Precedence Tables For F-Basic Operators

Precedence Table For Binary Arithmetic Operators

The following table contains the binary arithmetic operators grouped according to precedence. The higher precedence number indicates higher precedence. For a detailed discussion of precedence, see the section entitled 'Operator Precedence And Expressions' in Chapter 4.

<u>Operator Precedence Class</u>	<u>Precedence Number</u>
ASR, LSR, ASL, LSL, XOR	7
! (Immediate Assignment)	6
^ (Exponentiation)	5
/, \, *	4
-, +	3
<, >, =, <=, >=, <>	2
AND, MAX, MIN, IN, OR	1

This table groups F-Basic operators into precedence classes as a memory aid. Actually, within any given class, the operators have been listed with highest precedence starting on the left and lowest precedence ending on the right.

Precedence Table For Unary Arithmetic Operators

All unary operators have precedence higher than all binary operators.

Precedence Table For Pattern Matching Operators

<u>Operator Precedence Class</u>	<u>Precedence Number</u>
. (Conditional Assign.), ^ (Immediate Assign.)	3
* (Pattern Concatenation)	2
+ (Pattern Alternation)	1

The pattern match operator ? has precedence below all pattern

matching operations.

Precedence Table For Binary Operations On TEXT

<u>Operator Precedence Class</u>	<u>Precedence Number</u>
<, >, =	4
& (String Concatenation)	3
<=, >=, <>	2
MAX, MIN, IN	1

Index Of F-Basic Keywords, Functions, And Commands

This appendix provides an overview of F-Basic features. All mnemonics recognized by the FastCom compiler all presented in alphabetical order. With each entry is included a cross reference to a more detailed explanation and a 'telegraphic' description of the meaning or purpose of the item.

Compiler Directives

- &ANCHOR Chapter 19, Section 4
Changes pattern matching between anchored/unanchored mode.
- &DELIMITER Chapter 3, Section 3
Changes symbol which delimits text literals
- &EDITOR Chapter 1, Section 5
Changes editor loaded when syntax errors are discovered.
- &EVENT_CHECK Chapter 13, Section 2
Enables or disables event checking in main program.
- &FILESEP Chapter 7, Section 3
Changes symbol separating data items on a file.
- &QUICK Chapter 4, Section 1
Changes code generation for integer multiply and divide.
- &SEPARATOR Chapter 6, Section 1 Chapter 7, Section 3
Changes characters separating data items in keyboard input.
- &IGNORE Chapter 1, Section 5
Suppresses compilation of succeeding lines.
- &SYSLIB Chapter 16, Section 2
Allows high level access to Amiga ROM Kernel routines.
- &MENUWIDTH Chapter 12, Section 7
Changes default width of each menu item on menu bar.

F-Basic Language Feature Summary

? Chapter 2, Section 1
Line treated as comment if ? is first non-blank character.

{...} Chapter 5, Section 1
{ } pair encloses mnemonic or text statement labels.

[...] Chapter 5, Section 4
[] pair encloses CASE statement label.

[...] Chapter 6, Section 3
[] pair encloses format specification when used in PRINT.

ABORT Chapter 19, Section 5
System pattern that terminates pattern matching.

ABS Chapter 15, Section 1
Returns the absolute value of an INTEGER or REAL.

ACOS Chapter 15, Section 2
Returns the inverse cosine of a REAL.

ALLOCATE Chapter 15, Section 5
Allocates system memory block of a specified size.

ALLOCATE_WITH Chapter 15, Section 5
Allocates system memory block with specified attributes.

ANY Chapter 19, Section 5
System pattern that matches any character in a given list.

APPEND Chapter 10, Section 2
Adds a specified file to the end of the current file during compiles.

ARB Chapter 19, Section 5
System pattern that matches any string.

ASCII Chapter 15, Section 4
Changes a character to corresponding integer.

ASIN Chapter 15, Section 2
Returns the inverse sine of a REAL.

ATAN Chapter 15, Section 2
Returns the inverse tangent of a REAL.

AVAILMEM Chapter 15, Section 5
Returns the amount of system free memory.

BAL Chapter 19, Section 5

System pattern that matches a balanced string.

BASE	Chapter 15, Section 4
Converts text string to an integer using a specified base.	
BCHG	Chapter 15, Section 1
Complements specified bit of an INTEGER.	
BCLR	Chapter 15, Section 1
Sets specified bit of an INTEGER to zero.	
BETWEEN	Chapter 19, Section 5
Pattern matches any character between two given characters.	
BLOCK_CLOSE	Chapter 12, Section 6
Deallocates an image BLOCK structure.	
BLOCK_GET	Chapter 12, Section 6
Creates a BLOCK structure and saves rectangular image data.	
BLOCK_PUT	Chapter 12, Section 6
Copies image data from BLOCK structure to any open window.	
BLOCK_RESTORE	Chapter 12, Section 6
Copies image data from an array to a BLOCK structure.	
BLOCK_RPUT	Chapter 12, Section 6
Like BLOCK_PUT, except uses a relative positioning.	
BLOCK_SAVE	Chapter 12, Section 6
Copies image data from a BLOCK structure to an array.	
BREAK	Chapter 19, Section 5
Pattern matches any string not containing given characters.	
BSET	Chapter 15, Section 1
Sets specified bit of INTEGER to one.	
BTST	Chapter 15, Section 1
Returns the value of the specified bit in an INTEGER.	
BYTE	Chapter 3, Section 4
Eight bit INTEGER data type.	
CALL	Chapter 9, Section 5
Transfers program control to a SUBROUTINE.	
CHAR	Chapter 15, Section 4
Returns character corresponding to given INTEGER.	
CLOSE	Chapter 7, Section 5

- Closes a file.
- CLOSELIB** Chapter 15, Section 5 Chapter 16, Section 2
Closes specified Amiga ROM library.
- COLOR_AREA** Chapter 12, Section 5
Graphics function which defines polygon region for coloring.
- COLOR_AREAFILL** Chapter 12, Section 5
Graphics function which colors a polygon region.
- COLOR_BOX** Chapter 12, Section 4
Graphics function which draws a rectangular box.
- COLOR_BOXFILL** Chapter 12, Section 4
Graphics function which colors a rectangular box region.
- COLOR_DEFINE** Chapter 12, Section 4
Graphics function which sets the contents of a color register.
- COLOR_ELLIPSE** Chapter 12, Section 4
Graphics function which draws an ellipse.
- COLOR_FLOOD** Chapter 12, Section 5
Graphics function which fills any region with given color.
- COLOR_LINE** Chapter 12, Section 4
Graphics function which draws a line.
- COLOR_PATTERN** Chapter 12, Section 5
Graphics function which alters the default Amiga drawing patterns.
- COLOR_PENS** Chapter 12, Section 4
Graphics function which changes colors of drawing pens.
- COLOR_POINT** Chapter 12, Section 4
Graphics function which colors a pixel on the screen.
- COLOR_POLYDRAW** Chapter 12, Section 4
Graphics function which draws a polygon.
- COLOR_RAREA** Chapter 12, Section 5
Graphics function like **COLOR_AREA** using a relative location.
- COLOR_READ** Chapter 12, Section 4
Graphics function which returns the color of a pixel.
- COLOR_RELLIPSE** Chapter 12, Section 4
Gfxs function like **COLOR_ELLIPSE** using a relative location.

COLOR_RFLOOD Chapter 12, Section 5
Graphics function like COLOR_FLOOD using relative location.

COLOR_RPOINT Chapter 12, Section 4
Graphics function like COLOR_POINT using relative location.

COLOR_RREAD Chapter 12, Section 4
Graphics function like COLOR_READ using a relative location.

COLOR_RSCROLL Chapter 12, Section 6
Translates graphics objects inside a rectangle in a window.

COLOR_SCROLL Chapter 12, Section 6
Like COLOR_RSCROLL, except uses absolute positioning.

COMSTRING Chapter 15, Section 5
Returns command tail when user program is invoked from CLI.

CONSTANT Chapter 3, Section 6
Declares text mnemonics for literal data.

CONTINUE Chapter 5, Section 1
Equivalent to no operation on program line.

COS Chapter 15, Section 2
Returns the cosine of a REAL.

COSH Chapter 15, Section 2
Returns the hyperbolic cosine of a REAL.

COT Chapter 15, Section 2
Returns the cotangent of a REAL.

CSC Chapter 15, Section 2
Returns the cosecant of a REAL.

CURS_DN Chapter 6, Section 5
Moves the cursor down a specified number of rows.

CURS_INV Chapter 6, Section 5
Erases the cursor.

CURS_LE Chapter 6, Section 5
Moves the cursor to the left a given number of times.

CURS_LOC Chapter 6, Section 5
Moves the cursor to given row and column position.

CURS_RI Chapter 6, Section 5
Moves the cursor to the right a given number of times.

CURS_UP Chapter 6, Section 5
Moves the cursor up a specified number of rows.

CURS_VIS Chapter 6, Section 5
Restores presence of cursor on screen.

DATA Chapter 3, Section 5 Chapter 8, Section 3
Chapter 14, Section 6
Initializes the value of variables.

DEALLOCATE Chapter 15, Section 5
Returns a block of memory to system's free memory list.

DEC Chapter 15, Section 1
A quick subtraction of a number from a specified variable.

DELAY Chapter 15, Section 1
Causes specified delay in program.

DIGSTRING Chapter 15, Section 4
Converts an integer to a TEXT string.

DISPOSE_REC Chapter 14, Section 5
Deallocates dynamic memory for a RECORD type.

DOSEXECUTE Chapter 15, Section 5
Issues a DOS system command from F-Basic program.

DOSTIME Chapter 15, Section 5
Returns elapsed time.

ELSE Chapter 5, Section 3
Keyword in IF/THEN/ELSE/ENDIF structure.

ELSEIF Chapter 5, Section 3
Keyword in IF/THEN/ELSEIF/ENDIF structure.

END Chapter 2, Section 1
Keyword terminating program or subprogram.

END EVENT Chapter 13, Section 1
Terminates event processing block.

ENDCASE(S) Chapter 5, Section 4
Keyword which terminates WHEN (case) statement.

ENDIF Chapter 5, Section 3
Keyword in IF/THEN/ELSE/ENDIF structure.

ENDTYPE Chapter 14, Section 1

Keyword which terminates user defined TYPE declaration.

ENDWHILE Chapter 5, Section 6
Keyword which terminates a WHILE/ENDWHILE structure.

EOF= Chapter 7, Section 2
Transfers control when EOF encountered during INPUT #.

ERA_ALL Chapter 6, Section 5
Clears screen from cursor position to last row.

ERA_EOL Chapter 6, Section 5
Clears screen from cursor position to end of line.

ERA_LIN Chapter 6, Section 5
Clears screen in entire line containing the cursor.

EXCHANGE Chapter 15, Section 3
Swaps value of two variables of any data type.

EXEC Chapter 17, Section 4
Passes control to machine code program at absolute location.

EXIT Chapter 5, Section 5
Terminates execution of FOR loop when encountered.

EXP Chapter 15, Section 2
Returns the natural exponent of a REAL.

EVEN Chapter 15, Section 1
Returns one if given INTEGER is even and zero otherwise.

FAIL Chapter 19, Section 5
System pattern which causes failure of a pattern match.

FALSE Chapter 15, Section 1
System CONSTANT which is equivalent to zero.

FENCE Chapter 19, Section 5
System pattern limiting backtracking in pattern matching.

FILELENGTH Chapter 15, Section 5
Returns the length in bytes of a given disk file.

FILLCHAR Chapter 15, Section 3
Fills a string with a repeated text string.

FLOAT Chapter 15, Section 4
Converts INTEGER to REAL.

FOR Chapter 5, Section 5

Keyword in FOR/NEXT loop structure.

FRAC Chapter 15, Section 2
Returns fractional part of a REAL.

FUNCTION Chapter 9, Section 1 Chapter 14, Section 7
Keyword to declare a subprogram which returns a value.

GLOBAL Chapter 9, Section 2 Chapter 9, Section 3
Keyword indicates beginning of the global data section.

GOSUB Chapter 5, Section 2
Transfers control to a local subprogram.

GOSUB Chapter 19, Section 5
System pattern which transfers control to a local subprogram during pattern match operation.

GOTO Chapter 5, Section 1
Unconditional transfer of control to a given label.

GRETURN Chapter 9, Section 6
Return control from subprogram to calling program.

HEX Chapter 15, Section 4
Converts an INTEGER to a base 16 TEXT string.

HIWORD Chapter 15, Section 1
Returns upper 16 bits of a 32 bit INTEGER.

IABS Chapter 15, Section 1
Returns the absolute value of specified INTEGER.

IF...THEN Chapter 5, Section 3
Keywords in IF/THEN/ELSE/ENDIF structure.

INC Chapter 14, Section 7 Chapter 15, Section 1
A quick addition of a number to a specified variable.

INCLUDE Chapter 10, Section 1
Inserts given file after current position in a source file during compiles.

INPUT Chapter 6, Section 1
Reads data from keyboard.

INPUT # Chapter 7, Section 2
Reads data from file.

INS_LIN Chapter 6, Section 5
Inserts blank line above current cursor location.

INSERT Chapter 17, Section 3
 Inserts hex machine code in F-Basic program.

INT Chapter 15, Section 4
 Converts a REAL to an INTEGER.

INTEGER Chapter 3, Section 1
 Declares variables to be of type INTEGER.

INTSORT Chapter 15, Section 1
 Sorts an integer array.

ITEM_NUMBER Chapter 13, Section 3
 Returns menu item selected in MENU_SELECT event block.

IVAL Chapter 15, Section 4
 Converts TEXT string to base 10 INTEGER.

KEY_NUMBER Chapter 13, Section 3
 Returns ASCII value of key struck in INKEY event block.

LEN Chapter 19, Section 5
 System pattern which matches a given number of characters.

LENGTH Chapter 15, Section 3
 Returns non-blank length of TEXT string.

LEVEL Chapter 15, Section 3
 Locates substring at given level of TEXT string.

LN Chapter 15, Section 2
 Returns the natural logarithm of a REAL.

LOCAL Chapter 9, Section 2 Chapter 9, Section 3
 Declares start of LOCAL variable declarations.

LOCASE Chapter 15, Section 3
 Changes TEXT string to all lower case letters.

LOWORD Chapter 15, Section 1
 Returns the lower 16 bits of a 32 bit INTEGER.

LPOS Chapter 19, Section 5
 System pattern which matches a string starting a given number of characters from the left end.

LRETURN Chapter 5, Section 2
 Returns control from a local subprogram.

MAXINT Chapter 15, Section 1

Largest positive 32 bit INTEGER.

MAXREAL Chapter 15, Section 2
 Largest positive REAL number.

MENU Chapter 12, Section 7
 Creates user defined menus on menu bar.

MENU_CHECK Chapter 12, Section 7
 Renders a checkmark in a menu item.

MENU_CLOSE Chapter 12, Section 7
 Returns resources used in previous MENU statement to system.

MENU_DISABLE Chapter 12, Section 7
 Renders a menu or menu item unselectable.

MENU_ENABLE Chapter 12, Section 7
 Renders a menu or menu item selectable.

MENU_NUMBER Chapter 13, Section 3
 Returns menu title selected in MENU_SELECT event block.

MENU_OFF Chapter 12, Section 7
 Removes menu choices from menu bar.

MENU_ON Chapter 12, Section 7
 Displays menu choices on menu bar.

MINREAL Chapter 15, Section 2
 Smallest positive REAL number.

MOUSE_X1, MOUSE_Y1, MOUSE_X2, MOUSE_Y2 Chapter 13, Section 3
 Returns initial and final mouse pixel coordinates in SINGLE_CLICK event block.

NAME Chapter 19, Section 5
 System pattern which matches legal variable names.

NARRATE Chapter 11, Section 1
 Function which 'says' a phoneme equivalent string.

NEW_REC Chapter 14, Section 5
 Creates a new RECORD variable.

NEXT Chapter 5, Section 5
 Keyword in FOR/NEXT loop structure.

NIL Chapter 15, Section 1
 System constant equivalent to zero.

NOTANY Chapter 19, Section 5
 System pattern matches any character not in given list.

NULL Chapter 19, Section 5
 System pattern which matches the empty string.

NUMBER Chapter 19, Section 5
 System pattern which matches a string of digits.

ODD Chapter 15, Section 1
 Returns one if given INTEGER is odd and zero otherwise.

ON DOUBLE_CLICK EVENT Chapter 13, Section 1
 Processes left mouse button double click event.

ON INKEY EVENT Chapter 13, Section 1
 Processes single key input event.

ON MENU_SELECT EVENT Chapter 13, Section 1
 Processes menu selection event.

ON SINGLE_CLICK EVENT Chapter 13, Section 1
 Processes left mouse button single click event.

ON WINDOW_CLOSE EVENT Chapter 13, Section 1
 Processes window close gadget click event.

ON...GOTO Chapter 5, Section 8
 Keyword in computed GOTO structure.

ON...GOSUB Chapter 5, Section 8
 Keyword in computed GOSUB structure.

OPEN Chapter 7, Section 1
 Opens a disk file for input or output.

OPENA Chapter 7, Section 1
 Opens a disk file for output in append mode.

OPENLIB Chapter 15, Section 5 Chapter 16, Section 2
 Opens a specified Amiga ROM library.

OTHERWISE Chapter 5, Section 4
 Keyword in WHEN/OTHERWISE/ENDCASE structure.

PARAMETER Chapter 9, Section 4
 Declares beginning of PARAMETER declaration in a subprogram.

PATTERN Chapter 19, Section 1
 Declares variables to be of type PATTERN.

PI Chapter 15, Section 2
 System constant representing the mathematical constant pi.

PRINT Chapter 6, Section 2
 Affects screen output.

PRINT # Chapter 7, Section 3
 Affects file output.

PROGRAM Chapter 2, Section 1
 Keyword specifying the first line in a source program.

PTR_TO Chapter 14, Section 4
 Declares variable to be of type pointer.

RABS Chapter 15, Section 2
 Returns the absolute value of a REAL.

RANDOM Chapter 15, Section 1
 Returns a random 32 bit INTEGER.

RANDOMIZE Chapter 15, Section 1
 Initializes the random number generator in RANDOM.

RANDOMREAL Chapter 15, Section 2
 Returns a random REAL number between zero and one.

REAL Chapter 3, Section 2
 Declares variables to be of type REAL.

REM Chapter 2, Section 1
 Specifies that the program line is a comment statement.

REPEAT Chapter 5, Section 7
 Keyword in REPEAT/UNTIL structure.

REPLACE Chapter 15, Section 3
 Replaces each occurrence of string1 by string2 in string3.

RESTORE_A_REG Chapter 17, Section 1
 Restores A registers from the system stack.

RESTORE_REG Chapter 17, Section 1
 Restores all D and A registers from the system stack.

RETURN Chapter 5, Section 2
 Returns program control from a local subprogram.

RPOS Chapter 19, Section 5
 System pattern which matches a string starting a given number of characters from the right end.

RVAL Chapter 15, Section 4
Converts TEXT string to a REAL.

SAVE_A_REG Chapter 17, Section 1
Saves A registers on the system stack.

SAVE_REG Chapter 17, Section 1
Saves all D and A registers on the system stack.

SCR_BACK Chapter 6, Section 5
Changes the background color of the screen.

SCR_BEEP Chapter 6, Section 5
Produces a visual flash on the screen.

SCR_CLER Chapter 6, Section 5
Clears entire screen.

SCR_FORE Chapter 6, Section 5
Changes the foreground color of the screen.

SCR_STYL Chapter 6, Section 5
Changes the default text fonts on the screen.

SCREEN Chapter 12, Section 1
Graphics function which opens a custom screen.

SCREEN_BACK Chapter 12, Section 1
Graphics function moves specified screen behind all others.

SCREEN_CLOSE Chapter 12, Section 1
Graphics function which closes a custom screen.

SCREEN_FRONT Chapter 12, Section 1
Graphics function moves specified screen to front of others.

SEC Chapter 15, Section 2
Returns the secant of a REAL.

SETARRAY Chapter 8, Section 4
Changes the declared number of elements of an array.

SIN Chapter 15, Section 2
Returns the sine of a REAL.

SINH Chapter 15, Section 2
Returns the hyperbolic sine of a REAL.

SKIPFILE Chapter 7, Section 4
Moves file pointer to the end of a file.

SOUND Chapter 11, Section 3
Plays specified sound waves over Amiga sound channel.

SPACE Chapter 6, Section 4 Chapter 15, Section 3
Returns a TEXT string with the given number of blanks.

SPAN Chapter 19, Section 5
System pattern which matches string composed entirely of characters from a specified string.

SQR Chapter 15, Section 2
A quick operation which returns the square of a REAL.

SQRT Chapter 15, Section 2
A quick operation returning the square root of a REAL.

STOP Chapter 9, Section 6
Causes immediate termination when encountered.

SUBPROGRAM Chapter 9, Section 3
Keyword declaring the beginning of subprogram declarations.

SUBROUTINE Chapter 9, Section 1
Declares names of subprograms or first line of subroutine.

SUBSTRING Chapter 15, Section 3
Locates position of string1 in string2.

SUCCEED Chapter 19, Section 5
System pattern which always succeeds.

SWAP Chapter 15, Section 1
Exchanges upper and lower 16 bits of 32 bit INTEGER.

TAN Chapter 15, Section 2
Returns the tangent of a REAL.

TANH Chapter 15, Section 2
Returns the hyperbolic tangent of a REAL.

TEXT Chapter 3, Section 3
Declared variables to be of type TEXT.

TONE Chapter 11, Section 2
Produces sound of specified frequency.

TRANSLATE Chapter 11, Section 1
Converts TEXT string to phoneme equivalent string.

TRUE Chapter 15, Section 1

System constant equivalent to one.

TXTSORT Chapter 15, Section 3
Sorts a TEXT array.

TYPE...IS RECORD Chapter 14, Section 1
Begins declaration of user defined type.

UNOPTIMIZE Chapter 17, Section 1
Suppresses FAST variable storage in processor registers.

UNTIL Chapter 5, Section 7
Keyword in REPEAT/UNTIL structure.

UPCASE Chapter 15, Section 3
Converts TEXT string to all upper case characters.

VOICE Chapter 11, Section 1
Changes the characteristics of the voice in NARRATE.

WAVE Chapter 11, Section 3
Defines an audible waveform for use in SOUND/TONE.

WAVE_END Chapter 11, Section 3
Deallocates resources used by WAVE statement.

WHEN Chapter 5, Section 4
Keyword in WHEN/OTHERWISE/ENDCASE structure.

WHILE Chapter 5, Section 6
Keyword in WHILE/ENDWHILE structure.

WHILE...DO Chapter 5, Section 6
Optional form of WHILE statement.

WINDOW Chapter 12, Section 2
Graphics function which creates a user defined window.

WINDOW_ACTIVE Chapter 12, Section 2
Graphics function which brings a given window to the front of all others and activates it.

WINDOW_BACK Chapter 12, Section 2
Graphics function which places a given window behind others.

WINDOW_CLOSE Chapter 12, Section 2
Graphics function which removes a window from display and returns its resources to the system.

WINDOW_FRONT Chapter 12, Section 2
Graphics function places given window in front of others.

WINDOW_INFO Chapter 12, Section 2
Graphics function which returns data relating to current window's status.

WINDOW_MOVE Chapter 12, Section 2
Graphics function which repositions a window.

WINDOW_NUMBER Chapter 13, Section 3
Returns the window number in which the event occurs.

WINDOW_OUTPUT Chapter 12, Section 2
Graphics function which directs PRINT statement output to a specified window.

WINDOW_SIZE Chapter 12, Section 2
Graphics function which alters the dimensions of a given window.

WORD Chapter 3, Section 4
Declares variables to be 16 bit INTEGER.

FastCom Compile Time Error Messages

The FastCom compiler detects over 125 various error conditions that may exist in a user's program. The authors of the F-Basic language and the FastCom compiler have paid particular attention to the trapping of the kinds of programming errors that lead to significant debugging time if left undetected by the compiler. Each of the error messages is listed below.

Several of these error messages provide other pertinent information at the time the error is diagnosed. For example, Error #2, Illegal Identifier:, will also provide the offending variable name at the time the error is reported to the user.

Finally, many of the errors occur when the default limits of the FastCom compiler are exceeded. Such things as the number of allowed variable names, the number of subprograms, the number of record field names, and the number of user defined constants, etc. are limited by the compiler. Although the limits are extremely large, they may be exceeded in rare cases. The appendix titled 'Ordering Customized Versions Of The FastCom Compiler' provides the details concerning these limits and the ease with which a customized version of the compiler, with extended limits for the application at hand, may be obtained.

Error Message Number And Text:

- [1] Syntax Error In This Line
- [2] Illegal Identifier:
- [3] Illegal Location For This Type Of Declaration
- [4] END Expected
- [5] Character Label That Is Undefined Or Illegally Located In A FOR Loop:
- [6] PROGRAM Statement Must Begin Program
- [7] Illegal Array or Text Substring Reference
- [8] Undefined Array/Subroutine/Function/Library Reference:
- [9] Illegal Operator Used In This Line
- [10] Syntax Error In Format Description
- [11] Illegal Mixed Mode Operation In This Line
- [12] A Previously Declared Variable Was Found:
- [13] An Illegal Structure In a Pattern Was Found:
- [14] Mismatched Delimiters Found In Text Literal
- [15] Illegal Use of % Symbol or Too Many Text Literals In Line
- [16] Illegal Use Of # \$ Or ^ Operator On Non-INTEGER Pointer
- [17] Declarations Must Occur In The Order: <GLOBAL> PARAMETER LOCAL
- [18] Illegal 68000 Register Reference After :
- [19] Syntax Or TYPE Error In INC/DEC Argument
- [20] Illegal Subscript Or Length In This Declaration

[21] An Undefined Numeric Label In This Module Was Discovered:
 [22] An Undeclared SUBPROGRAM Was Discovered:
 [23] Illegal IF/WHILE/REPEAT Structure Encountered
 [24] Improperly Closed IF/WHILE/REPEAT Structure In This Module
 [25] Illegal FOR Loop Structure Encountered
 [26] Illegal WHEN Structure Encountered
 [27] Scalar Variables Must Be Declared In First 32K Of The Data Area
 [28] Optimized FAST Variables Have No Memory Address
 [29] Null Strings Are Undefined and Therefore Illegal
 [30] The Use of () in Concatenation is Redundant & Therefore Illegal
 [31] A File's Unit Number Must Be From 1 to 10
 [32] A FOR Loop Of Type INTEGER Must Have INTEGER Limits
 [33] A FOR Loop Index Must Be Of Type INTEGER Or REAL
 [34] Mixed Mode Multiple Equates Are Illegal
 [35] WHEN Expressions and Labels Must Be Integer or Text
 [36] A WHEN Expression And Its Labels Must Be Of The Same Type
 [37] The Number of GLOBAL Plus LOCAL Variables Is Excessive
 [38] The Number of String Literals In This Module Is Excessive
 [39] The Number of Labels In This Module Is Excessive
 [40] The Number of Forward GOTO/GOSUB References Is Excessive
 [41] The Number of Forward SUBPROGRAM References Is Excessive
 [42] The Number of SUBPROGRAMS Is Excessive
 [43] Improperly Closed WHEN Structure In This Module
 [44] EXIT Must Occur Inside a Loop
 [45] SUBPROGRAM Names Must Be Distinct From Variable Names
 [46] Syntax Error In This CONSTANT Declaration
 [47] CONSTANT Names Must Be Distinct From Declared Variables
 [48] The Number of INTEGER CONSTANTS Is Excessive
 [49] Syntax Error In This DATA Statement
 [50] The Array In A DATA Statement Must Be Of TYPE Integer Real Or Text
 [51] This DATA Statement Assigns Too Many Values To Its Object
 [52] An Undefined or Illegal CONSTANT In A Declaration Is :
 [53] DATA Statements Are Allowed Only In The Main Program
 [54] A Legal Variable Name For This DATA Statement Was Not Specified
 [55] The Number of Mnemonic Labels In This Module Is Excessive
 [56] DosExec Reports Inadequate FreeMem For Object Module Of This Size
 [57] Branch Out Of Range...Maximum Displacement Is 32K Bytes
 [58] Syntax Error Discovered In Command String
 [59] The Label On This Line Has Been Used Previously In This Module
 [60] Object Code Length Exceeds Current -O Option Size
 [61] The GRETURN Statement Is Inappropriate In The Main Program
 [62] Syntax Error In Declaration...
 [63] Length Of Text Literals In This Module Exceeds The -s Option Size
 [64] The Unary NOT Must Operate Only On Type Integer Or Logical
 [65] The Unary '-' Must Operate Only On Type Integer Or Real
 [66] Syntax Error In RECORD Reference Discovered
 [67] Attempt To Access A Field Of A Non-RECORD...Caused By The "."
 [68] An Illegal Field Name In A RECORD Reference Is:
 [69] A RECORD TYPE May Not Contain Itself As A Field

- [70] Nested TYPE Declarations Are Illegal
- [71] Syntax Error In TYPE Declaration Discovered
- [72] The Number Of User Defined TYPES In This Module Is Excessive
- [73] Illegal Mixed Mode In RECORD Equate/Input Discovered
- [74] Illegal Reference Discovered On Left Side Of The Assignment Equals
- [75] Illegal Field Width Specification Encountered
- [76] ENDTYPE Statement Missing In This Module
- [77] Array Index(es) And Substring Indices Must Be Of TYPE Integer
- [78] The STOP Statement Is Only Allowed In The Main Program
- [79] Arrays With Multiple Subscripts Are Not Allowed In Multi-Equates
- [80] TYPE REAL To Right => Only # Indirect Is Allowed On Left
- [81] The Maximum Depth Of Nested INCLUDE Files Has Been Exceeded
- [82] The Maximum Number Of Queued APPEND Files Has Been Reached
- [83] Branching From A FOR Loop Is Illegal - Use The EXIT Statement
- [84] Illegal Variable TYPE Discovered In This Line
- [85] Indirect Reference (^\$#) Cant Be Used With Variable TYPE On Right
- [86] The Length Of This CONSTANT Is Excessive
- [87] The Number Of Non-INTEGER CONSTANTS Is Excessive
- [88] Illegal TYPE For PTR TO To Point To
- [89] Maximum Number Of Self PTR TO Variables In This RECORD Exceeded
- [90] Parameter Discovered In This MACRO Call Has The Wrong TYPE
- [91] The Storage Requirement For This RECORD Is Excessive
- [92] An Illegal Argument In This SETARRAY Statement Was Discovered
- [93] The Argument Of DISPOSE_REC Must Be A PTR TO A RECORD TYPE
- [94] The Argument Of NEW_REC Must Be A RECORD TYPE
- [95] The Argument Of This CURS/SCR/DELAY Function Must Be Of TYPE INTEGER
- [96] Auto Inc/Dec Operation Is Allowed On TYPE INTEGER Data Only
- [97] Arrays Within RECORDS Must Be Singly Subscripted
- [98] Illegal Use Of The @ Operator Is This Expression
- [99] A Record With Array Fields Is Illegal In A DATA Statement
- [100] This Attempt To Move More Than 32K Of Data At One Time Is Illegal
- [101] TYPE Declarations May Not Occur In The PARAMETER Section
- [102] Array Equates w/ RECORD Or PTR_TO Parameters Are Not Permitted As
Their Length Is Unspecified
- [103] BYTE And WORD FUNCTIONS Are Not Allowed-Use TYPE INTEGER
- [104] TEXT/RECORD FUNCTIONS For A Line In This Module Exceed
32K-Memory Limit
- [105] The FUNCTION Name In This SUBPROGRAM Was Not Assigned A Value
- [106] Argument Of VOICE Function Must Be Singly Subscripted WORD Array
Name
- [107] The Number Of Subscripts Does Not Match Declared Number For:
- [108] The Total Number Of Field Names In All Defined TYPES Is Excessive
- [109] A FOR Loop With An Undeclared Index May Not Contain Nested Loops
- [110] Parameter Count Or Type Improper In Amiga System Library Call
- [111] Subscript In MultiArray Is Out Of Range...The Maximum Is 64K
- [112] Immediate Assignment Operator ! Is Only Valid On Integer Expressions
- [113] SCREEN or WINDOW or MENU Number Is Out Of Legal Range
- [114] Parameter Number Improper In SCREEN/WINDOW/COLOR/MENU Function Call
- [115] COLOR Function Pen Number Is Out Of Range

- [116] Concatenation Is Not Allowed In The WHEN Expression
- [117] Redefining A Previously Defined Data TYPE Is Illegal
- [118] Array Subscript Is Out Of Declared Bounds (During Compilation)
- [119] Subprogram-Previously Called Or Defined With Incorrect # Of Parameters
- [120] Subprogram-Previously Called Or Defined With Inconsistent Parameter Type
- [121] Incorrect # Of Parameters In A Call To An F-Basic Library Function
- [122] Inconsistent Parameter Type In A Call To An F-Basic Library Function
- [123] The EXCHANGE Operation Requires Two Objects Of The Same Type & Size
- [124] Optimized FAST Variables Can Not Be EXCHANGE'd
- [125] Field Reference Of A RECORD Parameter Is Not Allowed In EXCHANGE
- [126] BLOCK Number Is Out Of Range
- [127] The LineFeed Character Is Illegal In The &SEPARATOR Directive
- [128] Improper ON...EVENT Structure-Must Be In Main Program W/Out Nesting
- [129] EVENT Macros (WINDOW NUMBER etc.) Must Occur In ON...EVENT Structure
- [130] Branching Using GOTO/GOSUB Is Not Allowed In An ON...EVENT Structure

F-Basic Run Time Error Messages

During the execution of a user program, the F-Basic system detects error conditions that could not have been known at the time the program was compiled by the FastCom compiler. These are called 'run time' errors. When a run time error condition occurs, the F-Basic package alerts the user by displaying one of the following messages. Next an attempt to gracefully shut down all program resources, including returning memory to the system, closing open files, and closing open windows, screens and menus is made. This insures pleasant results in even the worst cases.

Many of the run time errors occur when a program inadvertently references a text string with a negative character length. Consider the following code:

```
PROGRAM TEST
INTEGER I,J,K
TEXT*10 STR
STR="ABCDEFGG"
I=5
J=3
K=LOCASE(STR(I:J))
END
```

Because the substring reference (I:J) of the TEXT variable STR produces an effective negative character length, run time error message #35 will result.

Error #1	An Error Code Wasn't Set By The Calling Routine
Error #2	An Overflow Occurred In Integer Multiply
Error #3	Illegal Character In Real Number INPUT Or RVAL
Error #4	Negative Text Length Discovered In CONCAT
Error #5	Non Digit Encountered In IVAL Library Function
Error #6	Negative Text Length Discovered In Text MOVER
Error #7	Argument In LN Library Function Less Than Zero
Error #8	Negative Argument In SQRT Library Function
Error #9	Negative Text Length In PATTERN Match Routine
Error #10	DOS System Error-Unable To PRINT To Named File
Error #11	Attempt To PRINT To UnOPENed File
Error #12	Attempt To INPUT Past Of End-Of-File
Error #13	DOS System Error-Attempt To Read File Failed
Error #14	Attempt To INPUT From UnOPENed File
Error #15	Attempt To CLOSE UnOPENed File
Error #16	DOS System Error-Requested Disk File Open Failed
Error #17	Attempt To OPEN UnCLOSEd File

Error #18	DOS System Error-Call To Open Standard Output Failed
Error #19	DOS System Error-Call To Open Standard Input Failed
Error #20	Negative Text Length Discovered In Text PRINT
Error #21	Non Digit Discovered In INTEGER INPUT Routine
Error #22	Negative Text Length In LENGTH Library Function
Error #23	Negative Text Length In LEVEL Library Function
Error #24	Illegal Character Or Negative Length In BASE
Error #25	Error Finding System Library For TRANSLATE
Error #26	Negative Text Length In IVAL Library Function
Error #27	Negative Text Length In Text Relational Routine
Error #28	Overflow In Argument-COS/SIN Library Functions
Error #29	Negative Text Length In FILLCHAR Library Function
Error #30	Negative Text Length In Text Input Routine
Error #31	Negative Text Length In Text Array=Text Scalar
Error #32	Argument Out Of Range In SPACE Library Function
Error #33	Error Finding System Library For NARRATE
Error #34	Negative Text Length In FILELENGTH Function
Error #35	Negative Text Length In UPCASE/LOCASE Functions
Error #36	Negative Text Length In RVAL Function
Error #37	Negative Text Length In TEXT MAX/MIN Functions
Error #38	Attempt To EXCHANGE Data Of A Different Size
Error #39	Division By Zero Detected
Error #40	Argument Out Of Range In ACOS Or ASIN Functions

Ordering Customized Versions Of The FastCom Compiler

Customized versions of the FastCom Compiler which extend the limits of various system capacities, such as the number of allowed variable names, statement labels per module, and total number of RECORD field names per program may be obtained at a nominal cost by submitting a request to :

Customized FastCom Versions
Delphi Noetic Systems, Inc.
P.O. Box 7722
Rapid City, SD 57709

The request must be submitted by a registered user and contain a precise statement of the required alterations desired. A reply detailing the costs involved will be forwarded to the user.

Considerations of size and speed of compilation have imposed reasonable limits on the system capacities of the FastCom Compiler for the average user. The commercially available version of FastCom was created with these 'average' constraints in mind. Although certain system capacities, like total object code size, concatenation buffer size, and string literal buffer size may be modified during invocation of the FastCom Compiler, it is recognized that in exceptional circumstances the other constraints may prove to be serious limitations. Thus, Delphi Noetic Systems, Inc. provides this service to overcome these difficulties.

Registering Commercial Stand Alone F-Basic Object Files

As explained in Chapter 1 of this manual, an F-Basic object module requires the presence of the copyrighted FastLib run time library to execute properly. For those F-Basic developers who produce commercial stand alone applications (for sale or without price), the registration form below is required to be returned to Delphi Noetic Systems, Inc. prior to distribution for each such application.

Registration Of F-Basic Commercial Stand Alone Applications

X	_____	X
	Company Name (if applicable)	
X	_____	X
	Name Of Individual To Contact	Day Time-(Area) Phone
X	_____	X
	Street Address (please do not use p.o. boxes)	
X	_____	X
	City, State, and Zip Code	
X	_____	X
	Application Program Name	
X	_____	X
	Brief Description Of Application-Attach A Letter If You Wish	
X	_____	X

RETURN THIS FORM TO :

Application Registrations
Delphi Noetic Systems, Inc.
P.O. Box 7722
Rapid City, SD 57709

Applications distributed without charge in the public domain, or those given to others without charge do not require a registration fee. Those sold for commercial distribution or classified as "shareware", or those with any other compensation due the programmer for their use must include a \$10.00 (Ten Dollar) registration fee with the above document.

F-Basic User's Manual Index

! operator 4.10
 & directives-general 1.8
 &DELIMITER directive 3.3
 &EDITOR directive 1.9
 &EVENT_CHECK directive 13.3
 &FILESEP directive 7.5
 &IGNORE directive 1.10
 &MENU_WIDTH directive 12.22
 &QUICK directive 4.2, 18.3
 &SEPARATOR directive 6.2, 7.5
 &SYSLIB directive 1.10, 16.2
 68000
 - assembly code 17.3
 - indirect reference 4.6, 18.3
 - registers 17.1
 & operator 4.6

A

ABORT pattern 19.9
 ABS function 15.3, 15.5
 ACOS function 15.5
 ALLOCATE function 15.11
 ALLOCATE_WITH function 15.11
 anchored mode 19.8
 AND operator 4.4
 ANY pattern 19.10
 APPEND STATEMENT 10.2
 FILL 12.16
 ARB pattern 19.9
 arrays
 - DATA statements 8.2
 - equates 8.2
 - multi-dimensional 8.5
 - one-dimensional 8.1
 - speed ups 18.2
 ASCII character set APP1
 ASCII function 15.9
 ASIN function 15.5
 ASL operator 4.5
 ASR operator 4.5
 assignments
 - immediate 4.10
 - multiple 4.10
 - statement 4.10
 ATAN function 15.5
 AVAILMEM function 15.11

B

BAL pattern 19.11
 BASE function 15.9
 bases other than 10 3.2, 15.9
 BCHG function 15.1
 BCLR function 15.1
 BETWEEN pattern 19.12
 BLOCK_CLOSE 12.18
 BLOCK_GET 12.17
 BLOCK_PUT 12.17
 BLOCK_RESTORE 12.19
 BLOCK_RPUT 12.17
 BLOCK_SAVE 12.18
 boxes-drawing 12.12
 BREAK pattern 19.11
 BSET function 15.1
 BTST function 15.1
 BYTE 3.5

C

CALL statement 9.5
 CHAR function 15.9
 circles-drawing 12.12
 CLOSE statement 7.6
 CLOSELIB function 15.12, 16.3
 color registers of the Amiga 12.9, APP 2
 COLOR_AREA 12.16
 COLOR_AREA
 COLOR_BOX 12.12
 COLOR_BOXFILL 12.12
 COLOR_DEFINE 12.9
 COLOR_ELLIPSE 12.12
 COLOR_FLOOD 12.15
 COLOR_LINE 12.11
 COLOR_PATTERN 12.14
 COLOR_PENS 12.10
 COLOR_POINT 12.10
 COLOR_POLYDRAW 12.13
 COLOR_RAREA 12.16
 COLOR_READ 12.11
 COLOR_RELLIPSE 12.12
 COLOR_RFLOOD 12.15
 COLOR_RPOINT 12.10
 COLOR_RREAD 12.11
 COLOR_RSCROLL 12.16
 COLOR_SCROLL 12.16
 comment statements 2.1
 compile time errors 1.8, APP 5

- compiler 1.1
- compiler directives
 - &DELIMITER 3.3
 - &EDITOR 1.8
 - &EVENT_CHECK 13.3
 - &FILESEP 7.5
 - &IGNORE 1.10
 - &MENUWIDTH 12.22
 - &QUICK 4.2, 18.3
 - &SEPARATOR 6.2, 7.5
 - &SYSLIB 1.10, 16.2
 - general 1.8
- compiler options 1.5
- COMSTRING function 15.12
- concatenation 4.6
- concatenation-default buffer length 1.6
- conditional assignment 19.6
- CONSTANT statement 3.7
- CONTINUE statement 5.1
- COS function 15.5
- COSH function 15.6
- COT function 15.5
- Countryman, N. ii
- CSC function 15.5
- cursor control 6.8
- CURS_DN 6.8
- CURS_INV 6.8
- CURS_LE 6.8
- CURS_LOC 6.8
- CURS_POS 6.8
- CURS_RI 6.8
- CURS_UP 6.8
- CURS_VIS 6.8
- case statement-see WHEN

D

- DATA statements
 - arrays 8.2
 - general 3.6
 - RECORDS 14.8
- DEALLOCATE function 15.11
- DEC
 - applied to pointers 14.9
 - function 15.2
 - subroutine 15.2
- declaration part 2.1, 9.1
- DELAY subroutine 15.2
- DIGSTRING function 15.10
- directory 1.4, 7.2, 10.2
- DISPOSE_REC 14.6
- DOSEXECUTE function 15.13
- DOSTIME function 15.13

- drawing modes of the Amiga 12.14
- dynamic
 - data objects 14.6
 - memory allocation 14.6

E

- E format 6.6
- ED editor 1.3
- ellipses-drawing 12.12
- end of file-see EOF
- END statement
 - general 2.1, 5.4
 - in subprograms 9.7
- EOF option 7.3
- ERA_ALL 6.8
- ERA_EOL 6.8
- ERA_LIN 6.8
- EVEN function 15.2
- event
 - active 13.2
 - checking 13.2
 - processing 13.1
- EXCHANGE subroutine 15.7
- EXEC 17.3
- executable part 2.1, 9.1
- EXIT statement 5.9
- EXP function 15.6
- expression 4.9

F

- F format 6.6
- F-Basic Developers Extension 8.1, 8.6
- FAIL pattern 19.10
- FALSE 15.2
- FAST variables 18.1
- FENCE pattern 19.14
- field
 - name 14.1
 - width 6.4
 - width overflow 6.6
- FILELENGTH function 15.14
- files
 - sequential 7.1
 - unit number 7.1
- FILLCHAR function 15.7
- FLOAT function 15.10
- FOR loop statement 5.8
- formatted output 6.3, 7.5
- FRAC function 15.5
- FUNCTION
 - general 9.1
 - user defined type 14.10

G

GLOBAL 9.2
go option (-G) 1.6
GOSUB pattern 19.13
GOSUB statement 5.3
GOTO statement 5.2
GRETURN statement 9.7
Griswold R.E. 19.1

H

HEX function 15.10
HIWORD function 15.3

I

IABS function 15.3
IEEE real number format 3.5
IF statement 5.4
immediate assignment 19.6
INC
 - applied to pointers 14.9
 - function 15.3
 - subroutine 15.3
INCLUDE statement 10.1
INKEY 13.7
INPUT
 - statement 6.1
 - to custom window 12.8
INPUT # statement 7.3
INSERT 17.3
INS_LIN 6.8
INT function 15.10
INTEGER 3.1, 3.2, 3.5
INTEGER arithmetic 4.2
interpreter 1.1
INTSORT function 15.3
ITEM_NUMBER 13.4
IVAL function 15.10

K

KEY_NUMBER 13.4

L

LEN pattern 19.11
LENGTH function 15.7
LEVEL function 15.7
line
 - drawing 12.11
 - numbers 2.1

literal

- general 3.1
- string buffer size 1.7

LN function 15.6
LOCAL 9.2
local subprogram 5.3
LOCASE function 15.8
LOWORD function 15.3
LPOS pattern 19.12
LRETURN statement 5.3
LSL operator 4.5
LSR operator 4.5

M

MAX operator 4.5
MAXINT 15.4
MAXREAL 15.6
memory allocation-dynamic 14.6
MENU 12.20
MENU_CHECK 12.23
MENU_CLOSE 12.23
MENU_DISABLE 12.22
MENU_ENABLE 12.22
MENU_NUMBER 13.4
MENU_OFF 12.23
MENU_ON 12.22
MID\$ operation 3.4
MIN operator 4.5
MINREAL 15.6
MOD operation-see remainder
mouse
 - current location 12.8
 - interaction with 13.1
MOUSE_X1 13.4
MOUSE_X2 13.4
MOUSE_Y1 13.4
MOUSE_Y2 13.4

N

NAME pattern 19.9
NARRATE 11.1
NEW_REC 14.6
NIL 15.4
NOT operator 4.6
NOTANY pattern 19.11
NULL pattern 19.10
NUMBER pattern 19.10

O

object
 - files 1.4
 - file default size 1.6

ODD function 15.4
 ON DOUBLE_CLICK EVENT 13.1
 ON INKEY EVENT 13.1
 ON MENU_SELECT EVENT 13.1
 ON SINGLE_CLICK EVENT 13.1
 ON WINDOW_CLOSE EVENT 13.1
 ON...GOSUB statement 5.12
 ON...GOTO statement 5.12
 OPEN statement 7.1
 OPENA statement 7.1
 OPENLIB function 15.12, 16.2
 operators

- address 4.6
- concatenation 4.6
- division 4.1
- exponentiation 4.1
- immediate assignment 4.10
- IN 8.4
- indirect reference 4.7
- logical 4.4
- masking 4.4
- multiplication 4.1
- precedence 4.8
- precedence table APP 3
- relationals 4.3
- remainder 4.1
- shift 4.5
- short circuit 4.4, 5.6
- subtraction 4.1
- unary 4.6

OR operator 4.4

OTHERWISE statement 4.7

P

PAR: device 7.2

parallel port-see PAR:

PARAMETERS

- arrays as 9.4
- passing by address 9.8, 14.10
- passing by value 9.8
- statement 9.2
- text as 9.4

path name 1.4, 7.2, 10.2

patterns

- alternation 19.1
- anchored mode 19.8
- concatenation 19.1
- conditional assignment 19.6
- definition 19.1
- immediate assignment 19.6
- matching 19.1, 19.4
- recursive 19.3
- system 19.8
- unanchored mode 19.8

PEEK 4.6

phoneme 11.1

phonetic equivalent string 11.1

PI 15.6

pointers

- general 14.1, 14.5, 18.2

- reference 14.5

POKE 4.6

polygons-drawing 12.13

precedence 4.8

precedence table APP 3

print option (-P) 1.7

PRINT

- statement 6.3

- to custom window 12.8

- USING 6.6

PRINT # statement 7.4

program lines

- continuation 2.2

- maximum length 2.2

- multiple statements 2.2

PROGRAM statement 2.1

prompt-input 6.1

R

RABS function 15.5

RAM: disk 1.10, 7.2

RANDOM function 15.4

RANDOMIZE subroutine 15.4

RANDOMREAL function 15.6

REAL

- internal format 3.5

- numbers 3.1, 3.2, 3.5

- scientific notation 3.2

- significant digits 3.3

record

- structures 14.1

- variable reference 14.4

recursion 9.9

register

- address 17.1

- data 17.1

- program counter 17.1

REM statement 2.1

remainder operation 4.1

REPEAT statement 5.11

REPLACE function 15.8

RESTORE_A_REG 17.2

RESTORE_REG 17.2

RETURN statement 5.3

RPOS pattern 19.12

run time errors 1.8, APP 6

RVAL function 15.10

S
 SAVE_A_REG 17.2
 SAVE_REG 17.2
 scalar 8.2
 SCREEN 12.2
 SCREEN_BACK 12.3
 SCREEN_CLOSE 12.3
 SCREEN_FRONT 12.3
 SCR_BACK 6.8
 SCR_BEEP 6.8
 SCR_CLER 6.8
 SCR_FORE 6.8
 SCR_STYL 6.8
 SEC function 15.5
 SER: device 7.2
 serial port-see SER:
 SETARRAY statement 8.4
 SIN function 15.5
 SINH function 15.6
 SKIPFILE statement 7.5
 SOUND 11.4
 source file 1.3
 SPACE function 6.7, 15.8
 SPAN pattern 19.11
 SQR function 15.6
 SQRT function 15.6
 stand alone object files
 - F-Basic 1.11
 - Registration 1.11, APP 8
 statement
 - label 5.1
 - number 5.1
 STOP statement 5.4
 structure 14.1
 subdirectory 1.4, 7.2, 10.2
 SUBPROGRAM declaration
 statement 9.3
 subprograms
 - general 9.1
 - global 9.7
 - invoking 9.5
 - recursive 9.9
 SUBROUTINE 9.1
 substring 3.4
 SUBSTRING function 15.8
 SUCCEED pattern 19.10
 SWAP function 15.4
 syntax 1.8

I

TAN function 15.5
 TANH function 15.6
 TEXT
 - as strings 3.3

- default length 3.4
- general 3.1, 3.3
- substring reference 3.4
- zero byte 16.1

TONE 11.3
 TRANSLATE 11.1
 TRUE 15.4
 TXTSORT function 15.9
 TYPE...IS RECORD 14.2
 types
 - system 14.1
 - user defined 14.1

U

unanchored mode 19.8
 unformatted output 6.3, 7.5
 UNOPTIMIZE 17.2, 18.2
 UPCASE function 15.8

V

variable names
 - case sensitivity 3.1
 - GLOBAL 9.2
 - legal names 3.1
 - LOCAL 9.2
 - PATTERN 19.1
 VARPTR 4.6
 VOICE 11.2

W

WAVE 11.4
 WAVE_END 11.4
 WHEN statement 5.6
 WHILE statement 5.10
 WINDOW 12.4
 WINDOW_ACTIVE 12.6
 WINDOW_BACK 12.6
 WINDOW_CLOSE 12.6
 WINDOW_FRONT 12.6
 WINDOW_INFO function 12.8
 WINDOW_MOVE 12.6
 WINDOW_NUMBER 13.4
 WINDOW_OUTPUT 12.6
 WINDOW_SIZE 12.6
 WORD 3.5

X

XOR operator 4.4

Z

zero byte termination 16.1

The F-Basic System User's Manual Errata

1. The manual's discussion of the INC function on page 15.3 should be clarified:

INC(<integer variable>)
INC(<integer variable>,<integer expression>)

INC may be used as a subroutine which increments a variable name representing an integer scalar or a singly subscripted integer array reference. <integer expression>, when present, determines the value to be added to <integer variable>. If <integer expression> is not present, INC adds one to <integer variable>. INC is equivalent to, but faster than, $\text{<integer variable>} = \text{<integer variable>} + \text{<integer expression>}$.

INC may be used as a function. In this case, the first parameter, <integer variable>, may be replaced with any integer expression. The function then returns the value of this expression plus the second parameter, <integer expression>, or plus one if the second parameter is absent.

INC, when applied to a variable of type PTR_TO INTEGER, PTR_TO REAL, or PTR_TO <RECORD type> increments the pointer by the length of the data object to which it points. For more details, see the chapter entitled 'RECORD Structures In F-Basic'.

2. The manual's discussion of the DEC function on page 15.2 should be clarified:

DEC(<integer variable>)
DEC(<integer variable>,<integer expression>)

DEC may be used as a subroutine which decrements a variable name representing an integer scalar or a singly subscripted integer array reference. <integer expression>, when present, determines the value to be subtracted from <integer variable>. If <integer expression> is not present, DEC subtracts one from <integer variable>. DEC is equivalent to, but faster than, $\text{<integer variable>} = \text{<integer variable>} - \text{<integer expression>}$.

DEC may be used as a function. In this case, the first parameter, <integer variable>, may be replaced with any integer expression. The function then returns the value of this expression minus the second parameter, <integer expression>, or minus one if the second parameter is absent.

DEC, when applied to a variable of type PTR_TO INTEGER, PTR_TO REAL, or PTR_TO <RECORD type> decrements the pointer by the length of the data object to which it points. For more details, see the chapter entitled 'RECORD Structures In F-Basic'.

3. The manual should discuss the RDEC function on page 15.6

RDEC(<real variable>)

<over>

RDEC, when used as a subroutine, decrements a real scalar by 1.0. Note that unlike the DEC subroutine for integers, singly subscripted array references are not permitted as a parameter. RDEC is equivalent to, but faster than, `<real variable>=<real variable>-1.0`.

RDEC, when used as a function, accepts any real expression in place of the parameter `<real variable>`, and returns the value of that expression minus 1.0.

4. The manual should discuss the RINC function on page 15.6

`RINC(<real variable>)`

RINC, when used as a subroutine, increments a real scalar by 1.0. Note that unlike the INC subroutine for integers, singly subscripted array references are not permitted as a parameter. RINC is equivalent to, but faster than, `<real variable>=<real variable>+1.0`.

RINC, when used as a function, accepts any real expression in place of the parameter `<real variable>`, and returns the value of that expression plus 1.0.

5. The CHAR function (discussed on page 15.10) actually accepts an integer argument in the range 1 to 255. Thus, the extended character set of the Amiga can, in fact, be accessed in this fashion.
6. The distinction between `?` and `REM` for program lines with multiple statements separated by semicolons (discussed on page 2.1) is more precisely given as follows. On such a line, a leading `?` causes the compiler to ignore the entire line, while `REM` comments out only that portion of the line until the next semicolon or the end of the line is encountered. This makes each useful depending upon the circumstances.
7. Chapter 7's discussion of the `PRINT#` and `INPUT#` statements should caution against alternating between these two commands as internal F-Basic file i/o buffering makes the results unpredictable and undesirable. It is always acceptable, however, to `INPUT#` to the end of a file, or use the `SKIPFILE` command to reach the end of a file, and then write to it using the `PRINT#` statement.
8. Chapter 9's discussion of the `FUNCTION` subprogram should include the requirement that the `FUNCTION` name must be assigned a value within the body of the subprogram. Failure to do so produces a compile time error.
9. The discussion of the `COMSTRING` function on page 15.13 should read: The `<result>` returned is always the integer 1. If the command tail was not utilized or left blank, then the value of `LENGTH(<text variable>)` will be zero after returning from the call to `COMSTRING`.
10. Chapter 12, Section 2. The `WINDOW_INFO` function should read: `WINDOW_INFO(8)` returns the Left Edge of the window, while `WINDOW_INFO(9)` returns the TopEdge of the window.

The F-Basic System User's Manual Additions

Two new arithmetic library functions, SGN and RSGN, are now available. The syntax is:

```
<Integer Result>=SGN(<Integer Parameter>)  
<Real Result> = RSGN(<Real Parameter>)
```

These functions simulate the mathematical sign function, which returns +1 if the parameter is positive, -1 if the parameter is negative, and 0 if the parameter is exactly zero.

Graphics Functions:

1. The value of Flags in the WINDOW # statement, discussed on page 12.5 in the third paragraph, should also include a value of 32. This value of Flags specifies a BORDERLESS window. Choosing this value indicates that a window without the standard double line border is desired.

2. The syntax of the WINDOW, MENU, BLOCK, COLOR, and SCREEN statements and functions has been expanded to permit window numbers, menu numbers, block numbers, pen numbers, and screen numbers to be specified as either integer literals or integer scalar variables. For example,

```
WindPtr=WINDOW #K (...list of window parameters...)  
or COLOR_DEFINE #Red (...list of color parameters...)
```

are now supported provided that K and Red are integer variables or constants whose value lies in the appropriate range. Because of this change, error checking of the range of these values by the compiler is no longer possible.

The MENU_ENABLE, MENU_DISABLE, and MENU_CHECK statements do not support this extension due to the complex meaning associated with their menu number.

3. The functions COLOR_POINT, COLOR_RPOINT, COLOR_LINE, COLOR_BOX, COLOR_BOXFILL, COLOR_ELLIPSE, COLOR_RELLIPSE, COLOR_POLYDRAW, COLOR_FLOOD, and COLOR_RFLOOD have each been speeded up by factors ranging from 2 to 6 over the original release.

4. A new library function, INCHAR, provides a single character input (without echo) capability. (The reader is assumed to be familiar with Chapter 12, Section 3 of the F-Basic System User's Manual). The syntax is:

Here,
F-Basic
char

<TEXT variable of declared length 1>=INCHAR()

The INCHAR function will wait for a single keystroke and return this text value in the <TEXT variable>. The input is accepted from the Current Output Window, provided a custom F-Basic window has been opened.

INCHAR does not provide a question mark prompt, nor does it echo the input character. It is also not available in the default AmigaDOS CLI window, and will return a zero byte character if used in that context.

5. A new graphics command, SLEEP, has been added. The syntax is simply:

SLEEP

Issuing the SLEEP command causes the F-Basic system to temporarily suspend execution until an input event is sensed. The proper use of the command involves opening one (or more) custom window, designating the input event(s) you are interested in trapping with the ON...EVENT/END EVENT structure, and then issuing a SLEEP directive. SLEEP utilizes the non-busy waiting feature of the Amiga's multi-tasking environment.

6. A new graphics command WINDOW_EVENT_ENABLE has been added. The syntax is simply:

WINDOW_EVENT_ENABLE

Recall that an ON...EVENT block automatically changes all currently open windows so that they may respond to the event specified in the block. However, windows that are opened after the ON...EVENT block has been encountered for the first time are not able to report the event contained in the block because they have not been changed in this manner. The command WINDOW_EVENT_ENABLE should be issued after opening any window past the point where the ON...EVENT block was encountered during execution. This command will cause the newly opened window to be changed in the ways necessary to report events as they occur.

File I/O

1. Two library functions, READBYTES and WRITEBYTES, provide raw character i/o from files. (The reader is assumed to be familiar with Chapter 7 of the F-Basic System User's Manual.) Their syntax is:

<integer result>=READBYTES(<unit number>,<buffer>,<number of bytes>)
and <integer result>=WRITEBYTES(<unit number>,<buffer>,<number of bytes>)

F-Basic Object Modules And The "FastLib" File

1. The Stand-Alone Object File capabilities of the F-Basic system, discussed in Chapter 1-Section 7, have been expanded. The accessory file "FastLib" needed by all F-Basic generated applications may now reside in the s directory of the sys: drive. Thus, if the file "FastLib" is not found in the current directory during execution, a search is made for the file "sys:s/FastLib" before the run-time message "Unable To Open File FastLib...It Must Be Present" is given by the F-Basic object program.

Delphi Noetic Systems, Inc.

=====

F-BASIC SOFTWARE LICENSE AGREEMENT

=====

PLEASE READ this statement before opening the disk package. By opening the package you show your agreement to and acceptance of these terms and conditions. If you do not agree to them, you should promptly return the entire contents of this package, unopened, to Delphi Noetic Systems, Inc., and your money will be refunded. Your right to return this product for a refund for this reason expires seven days after purchase.

1. Copyright - This program and its documentation are copyrighted. You may not copy or reproduce all or any part of the program, except to make a single archival copy of the software for the sole purpose of backing up the software and protecting your investment from loss. You must reproduce and include the copyright notice of Delphi Noetic Systems, Inc. on the backup copy. You may not use, copy, modify, or transfer the program or its documentation except as expressly provided in this agreement.
2. License - As consideration for your payment, you are hereby granted a non-transferable LICENSE to use the software subject to the restrictions and limitations contained in this agreement. In the event that you breach this agreement, Delphi Noetic Systems, Inc. may terminate this license. Upon such termination, you agree to send all copies of the software and documentation immediately to Delphi Noetic Systems, Inc. Additionally, you shall be liable for any and all damages suffered as a result of your violation. The LICENSE granted to you is effective until terminated.
3. Hardware - You may use the program only on a single computer owned or operated by you. You may transfer the program from one computer to another which you own or operate provided that you do not permit the use of the program on more than one computer at a time. You may not provide use of the software in a network or time sharing arrangement to end users who are not individually licensed by Delphi Noetic Systems, Inc. You may not distribute copies of the program or accompanying documentation to others. You may not modify, translate, or reverse engineer the program.
4. Compiled Programs - Programs that you write in the F-Basic language and compile with the FastCom compiler may be given away to others, made a part of "public domain" libraries, or sold provided the Registration of F-Basic Source and Object Modules Document, contained in Appendix VIII of the accompanying documentation, is returned to Delphi Noetic Systems, Inc., with the appropriate fee before such distribution.

5. Program Disclaimer - THE PROGRAM IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE PROGRAM IS ASSUMED BY YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU (AND NOT Delphi Noetic Systems, Inc. OR ITS DEALERS) ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING REPAIR OR CORRECTION. FURTHER, Delphi Noetic Systems, Inc. DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING THE USE OF, OR THE RESULTS OF THE USE OF, THE PROGRAM IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, CURRENTNESS, OR OTHERWISE; AND YOU RELY ON THE PROGRAM AND RESULTS SOLELY AT YOUR OWN RISK.
6. Diskette Limited Warranty - Delphi Noetic Systems, inc. WARRANTS TO THE ORIGINAL LICENSEE THAT THE DISKETTE(S) ON WHICH THE PROGRAM IS RECORDED SHALL BE FREE FROM DEFECTS IN MATERIAL AND WORKMANSHIP ONLY FOR A PERIOD OF NINETY (90) DAYS FROM THE DATE OF THE ORIGINAL PURCHASE. IF A DEFECT COVERED BY THIS WARRANTY OCCURS DURING THIS 90-DAY WARRANTY PERIOD, AND IT IS RETURNED TO THE DEALER FROM WHICH IT WAS PURCHASED NOT LATER THAN FIVE (5) DAYS AFTER THE END OF SUCH 90-DAY WARRANTY PERIOD, THE DEALER SHALL, AT THE DEALER'S OPTION, EITHER REPAIR OR REPLACE THE DISKETTE. THIS WARRANTY IS IN LIEU OF ALL OTHER EXPRESS OR STATUTORY WARRANTIES, AND THE DURATION OF ANY IMPLIED WARRANTY, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, IS HEREBY LIMITED TO SAID NINETY (90) DAY PERIOD. Delphi Noetic Systems, Inc.'s LIABILITY IS LIMITED SOLELY TO THE REPAIR OR REPLACEMENT OF THE DEFECTIVE PRODUCT, IN ITS SOLE DISCRETION, AND SHALL NOT IN ANY EVENT INCLUDE DAMAGES FOR LOSS OF USE OR LOSS OF ANTICIPATED PROFIT OR REVENUE, EXPENSES, OR DAMAGES, INCLUDING WITHOUT LIMITATION ANY DATA OR INFORMATION WHICH MAY BE LOST OR RENDERED INACCURATE, EVEN IF Delphi Noetic Systems, Inc. HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. ADDITIONALLY, Delphi Noetic Systems, Inc. IS NOT RESPONSIBLE FOR THE DAMAGES OR COSTS INCURRED IN CONNECTION WITH OBTAINING SUBSTITUTE SOFTWARE, CLAIMS BY OTHERS, INCONVENIENCE, OR SIMILAR COSTS.

Some states do not allow a limitation on how long an implied warranty lasts, so the above limitation may not apply to you. Some states do not allow the exclusion or limitation of incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which may vary from state to state. In no event will the liability of Delphi Noetic Systems, Inc. for any damages to you or any other person ever exceed the price paid for the software, regardless of any form of the claim.

7. Entire Agreement - This agreement represents the entire agreement between the parties hereto with respect to the subject matter contained herein and supersedes any previous oral or written communications, representations, understandings, or agreements. The terms of this agreement may be modified only in writing signed by both parties hereto.
8. Governing Law - This software license agreement shall be interpreted and governed by the laws of the state of South Dakota.

More Amiga Products...

Watch For These Upcoming Amiga Products
From Delphi Noetic Systems, Inc.

Registered users of the F-Basic System will receive the most timely notices of the shipping dates of the following new Amiga products from Delphi Noetic Systems, Inc. Additionally, remarkable price reductions on our continuing line of products will be offered to all of the members of the registered F-Basic development family. Please, return the enclosed product purchase registration card to guarantee that you are included in these announcements.

The officers of Delphi Noetic Systems, Inc. have made a firm commitment to the prompt delivery of advertised software. Unlike some Amiga software vendors, when we say we're shipping, we are shipping!

Now In Gamma Testing:

The F-Basic Source Level Debugger And Disassembler, including:

1. A fully windowed Intuition interface for the easiest Amiga development environment and best productivity from your computing time.
2. An extensive list of features. Several memory display modes, memory searches, source level and assembly level breakpoints, single step and continuous trace, allocate memory, mini-assembler, and log files.
3. A working knowledge of 68000 assembly not necessary to debug F-Basic source programs.
4. The best Development Extension we can offer to our many F-Basic developers.

<over>

Now In Alpha And Beta Testing:

An enhanced FastCom compiler, including support for:

1. Extended double precision REALs with the same F-Basic passion for speed and accuracy evident in our single precision format.
2. Random Access Files.
3. User Defined Operations On RECORD Structures (Define matrix operations, for example).
4. Complex Numbers.
5. Direct utilization of the enhanced 68020/68881 addressing modes when these hardware upgrades are present.
6. In Line Assembly Language Mnemonic Instructions.
7. Direct High Level Animation Routines.
8. Direct IFF file support.

Now In Product Development:

The F-Basic Logic Programming Extension, including the best of the features of Lisp and PROLOG, in the familiar F-Basic environment.

Delphi Noetic Systems, Inc.

=====

Contents of the F-Basic System Sample Programs Disk

=====

INTRODUCTION

The enclosed F-Basic System Sample Programs disk contains over 65 source programs written in the F-Basic language. By examining, compiling, and running those which use the specific features of the language that you are interested in, the authors hope to acquaint you with the power, speed, and flexibility of the F-Basic System.

If you have at least a two drive system (one additional external drive df1:), the easiest way to compile and run the sample programs is:

1. Read Chapter 1 of the F-Basic System User's Manual to acquaint yourself with the FastCom compiler's operation.
2. Boot your Amiga with the F-Basic System disk in the internal drive and the F-Basic System Sample Programs disk in the external drive.
3. To compile a sample program (for example GfxsTest24), type:

```
l>FB df1:GfxsTest24 test.bin
```

This will compile the test program and place the object code in the file called "test.bin" of the internal drive.

4. To run the program, type:

```
l>test.bin
```

If you have a single drive system, the sample programs that you desire to compile and execute should be first moved to the System Disk. Alternatively, for those with extended memory systems, the FastCom compiler can be moved to the RAM: disk, as described in Chapter 1 of the F-Basic System User's Manual. This will enable you to utilize the internal drive to hold the F-Basic System Sample Programs disk.

THE SAMPLE PROGRAMS

1. CribbageCombos1 - Asks for the number of cards in the user's hand and the target number. It then recursively computes the number of ways of permuting the cards to get the target number.

Example Input: 7,8 (7 cards and 8 is the target)
Cards Input One To A Line: 1,2,3,4,5,6,7,
Answer: There are 5 ways to get 8.

2. CribbageCombos2 - Same as CribbageCombos1, except an iterative, rather than recursive, solution is demonstrated.
3. CursorControl - Demonstrates the cursor control and line and blank features of F-Basic. No user input requested - simply compile, run, and watch.
4. GfxsTest1 - Demonstrates high level custom windows from F-Basic. Opens a custom window and draws 3 lines therein. Window resides in the default Workbench screen.
5. GfxsTest2 - This is a repeat of GfxsTest1, except the window is opened in a custom low resolution screen.
6. GfxsTest3 - Demonstrates user defined colors, line drawing, color reading pixel by pixel, and color complementing in F-Basic. Uses a custom window defined in a custom screen.
7. GfxsTest4 - Demonstrates the use of the F-Basic COLOR_POLYDRAW function by drawing repeated octagons in a custom window defined in a custom screen.
8. GfxsTest5 - Demonstrates patterned lines and circle drawing in F-Basic. Uses the COLOR_PATTERN, COLOR_ELLIPSE, and COLOR_FLOOD statements.
9. GfxsTest6 - Demonstrates user defined colors, drawing patterned lines, and changing the drawing mode through the use of F-Basic's COLOR_PATTERN statement.
10. GfxsTest7 - Draws a circle, and then demonstrates F-Basic's COLOR_FLOOD statement to fill the circle and its background with different colors.
11. GfxsTest8 - Demonstrates F-Basic's COLOR_ELLIPSE and COLOR_RELLIPSE statements by drawing cocentric circles in random colors in a custom window and screen.
12. GfxsTest9 - Demonstrates F-Basic's COLOR_ELLIPSE, COLOR_RELLIPSE, and COLOR_FLOOD statements by drawing and filling cocentric circles in random colors in a custom window and screen.
13. GfxsTest10 - Like GfxsTest4 with the moving octagons, except here they are drawn with and filled by the COLOR_AREA and COLOR_AREAFILL statements of F-Basic. Random numbers from the F-Basic RANDOM facility are used to determine the fill colors used.
14. GfxsTest11 - Demonstrates patterned fills in all 4 distinct drawing modes on the Amiga - JAM1, JAM2, COMPLEMENT, and INVERSE VIDEO. Outlines a random box in a random color and then fills with a user-defined pattern via F-Basic's COLOR_PATTERN statement.
15. GfxsTest12 - Beautiful demo of filling a circle, triangle, and rectangle with a randomly selected fill pattern. Includes the use of F-Basic's COLOR_PATTERN, COLOR_AREA, and COLOR_AREAFILL statements.

16. GfxsTest13 - Like the sample titled GfxsTest12, except the multiple color patterned fill capabilities of the Amiga, as accessed through F-Basic's COLOR_PATTERN statement, are demonstrated.
17. GfxsTest14 - Lines and shapes are drawn and filled with random patterns using the COLOR_PATTERN statement. Includes the use of COLOR_DEFINE, COLOR_ELLIPSE, COLOR_AREA, COLOR_LINE, and COLOR_BOXFILL.
18. GfxsTest15 - Like the program called GfxsTest14, except again multiple color patterns are employed.
19. GfxsTest16 - Utilizes the F-Basic COLOR_SCROLL statement to simulate the movement of the Space Invaders video game.
20. GfxsTest17 - Similar to GfxsTest16, except the relative scrolling capabilities of the Amiga, as accessed through F-Basic's COLOR_RSCROLL statement, are utilized.
21. GfxsTest18 - Demonstrates F-Basic's high level menu interface. Periodically, touch the right mouse button and drag down the menu while this one executes. You will notice all the program directed changes in the menu interface.
22. GfxsTest19 - Demonstrates F-Basic's high level BLOCK statements. These allow the programmer to quickly capture a graphics image on the screen and replicate it throughout a window, or save it in an array or file for later use.
23. GfxsTest20 - Further illustrates F-Basic's BLOCK facility by transferring an image to a second custom window. During execution, click on the screen depth arrangement gadget to view the window from which the original image was copied.
24. GfxsTest21 - Same program as GfxsTest19, except the image is copied first to an array for permanent storage, then restored from the array and duplicated throughout the window.
25. GfxsTest22 - Same program as GfxsTest20, except as in GfxsTest21, the image is transferred between two windows through the use of an intermediate array for the possible permanent storage of the image.
26. GfxsTest23 - Demonstrates F-Basic's WINDOW statements which allow the programmer to move and resize windows as well as arrange their locations relative to each other. A custom screen and four custom windows are demonstrated.
27. GfxsTest24 - Program opens eight custom windows simultaneously to demonstrate the 32 default colors of the Amiga, the F-Basic version of the Boxfill demo, the F-Basic version of the random Box demo, the F-Basic version of the Lines demo, the F-Basic version of the Dots demo, and the F-Basic version of the Ellipses demo.
28. GfxsTest25 - Like GfxsTest19, except the relative mode of the BLOCK statement is utilized.

29. GfxsTest26 - Illustrates how to open and receive input from a requester using Intuition's Auto Requester facility coupled with F-Basic's RECORD structures.
30. GfxsTest27 - Program demonstrates F-Basic's Event Trapping facilities. Two windows are opened, and each event in each window is reported to the user. The events possible in each window are a WINDOW_CLOSE event, an INKEY (keyboard strike) event, a MENU_SELECT event, a SINGLE_CLICK event, and finally, a DOUBLE_CLICK event.

To use, perform an event (one of the above) in the desired window and watch for the program to report the occurrence of the event to you.

31. Hanoi - Program outputs a sequence of instructions to solve the Towers of Hanoi problem.

Example Input: Disks To Be Moved? 3
 Needle To Move From? 1
 Needle To Move To? 3
 Needle To Use AS Temporary Storage? 2

32. IntArith1 - Demonstrates integer arithmetic operations in F-Basic, both on optimized FAST and unoptimized variables. Correct output: 318, 187200, 5830380, 2, 582.
33. IntArith2 - Further tests of integer arithmetic operations in F-Basic. Additionally, nested FOR loops are utilized. Correct output: 100125, 99, 1177, 15210, 235350.
34. IntArith3 - Integer arithmetic on integer arrays. Correct output: 99, 420.
35. IntArith4 - Tests integer arithmetic on arrays by finding and outputting all primes below 100.
36. Jumble1 - Program simulates the word "shuffling" game Jumble. Enter a string of letters, and all the permutations of those letters are displayed. Note that duplicate permutations are not output.
37. Jumble2 - Like Jumble1, except a more compact recursive algorithm is used in the program.

The next seven sample programs illustrate the ease with which C programs that interact directly with the ROM Kernel of the Amiga can be programmed in F-Basic. The RECORD structures, file INCLUDE and APPEND, and &SYSLIB interface of F-Basic are demonstrated. NOTE: These programs assume that the "finclude" directory of the F-Basic System boot disk is present. Many include files are located here.

38. LikeC1 - This program defines MenuItem and MenuText RECORDS to attach a menu bar to a custom window, delay for 10 seconds, and then close down. Touch the right mouse button during execution to see the menu bar. Contrast this approach with F-Basic's high level interface, as demonstrated in GfxsTest18.

39. LikeC2 - This program uses calls to the DOS library to examine the information on the disk about any file. As an example, input "FB", and the information about the FastCom compiler (file FB) is displayed.
40. LikeC3 - This program directly accesses the Amiga's Sound library functions and audio device to play a sine wave with a variable number of points a variable number of times. Example input: 1000, 150, 20.
41. LikeC4 - This program uses direct calls to the translator library and the narrator device to "speak" the contents of a file. Enter a file name, and it reads the file to you. Example input: FBasic.Example.
42. LikeC5 - This program directly uses the blitter to move an orange box from one area of a custom window to another.
43. LikeC6 - Like the program LikeC5, except the blitter is directly employed to move an image from one custom window to another.
44. LikeC7 - This program uses direct ROM library calls to sense input events after opening 2 windows and attaching 2 menus. The events detected are Menu Selection, Mouse Clicks, and Close Window Clicks. After each input event, the user is given the option to either terminate or continue.
45. LinkedList - This program demonstrates F-Basic RECORD and PTR_TO (pointer) structures by constructing and maintaining a linked list of integers. The operations supported include adding to, deleting from, and erasing the linked list, as well as stopping the program. The command "erase" should be executed before "stopping" to guarantee that all free memory is returned to the operating system.
46. MixedArith1 - A demonstration of F-Basic's mixed mode arithmetic is presented. The correct output is listed in the program's header block.
47. MixedArith2 - A further demonstration of F-Basic's mixed mode arithmetic capabilities. The correct output is again listed in the program's header block.
48. RealArray1 - This program demonstrates F-Basic's real array capabilities by sorting a reverse-filled array. The correct output is 1.0 through 20.0
49. Sound1 - This program demonstrates F-Basic's high level sound facilities. The musical scale is played five times. First the F-Basic TONE function is used. Next, using the WAVE statement, the scale is played for a sawtooth wave, then a square wave, then a sine wave, and finally, for a "noise" wave.
50. TextFunctions - Demonstrates F-Basic's ability to support functions of type TEXT. Concatenation and substring references of them are also included. The correct output is listed in the program header block.

51. TextSort - This program demonstrates F-Basic's TXTSORT function. It randomly fills a character array and then sorts the elements. An input for the number of elements desired by the user is accepted (maximum = 100 elements for this program).

Most of the following timing tests were taken from Amiga publications where other language systems were discussed. The published programs were converted to F-Basic and then timed using the system's timer, as accessed by F-Basic's DOSTIME function. Where applicable, a reference to the original article is given in the header block of the program.

52. TimingTest1 - A prime program published in Amazing Computing, Volume 1, #6, page 6.
53. TimingTest2 - Another prime program published in Amazing Computing, Volume 1, #9, page 114.
54. TimingTest3 - A sieve program using a WHILE control structure. Published in Amazing Computing, Volume 1, #9, page 114. Ten passes are timed.
55. TimingTest4 - Another version of the sieve program using a FOR loop control structure. Published in Amazing Computing, Volume 1, #9, page 114. Ten passes are timed.
56. TimingTest5 - An optimized sieve program demonstrating F-Basic's address (@) and indirect reference (^) operators. This is a slight modification from programs TimingTest3 and TimingTest4, and is timed for 10 passes through the algorithm.
57. TimingTest6 - Savage benchmark as published in Amazing Computing, Volume 2, #4, page 8.
58. TimingTest7 - Benchmark timing test displayed in Amazing Computing, Volume 1, #9, page 79.
59. TimingTest8 - Benchmark timing test as shown in Amazing Computing, Volume 1, #9, page 79, except F-Basic's REAL FOR loops are utilized.
60. TimingTest9 - Same benchmark timing test as TimingTest8, except that F-Basic's INTEGER FOR loops are used.
61. TimingTest10 - Float timing test displayed in Amazing Computing, Volume 2, #8, page 75.
62. TimingTest11 - BYTE Calculations program, as presented in Amazing Computing, Volume 2, #8, page 75.
63. TimingTest12 - WriteToDisk timing test presented in BYTE, Fall, 1985, page 200.
64. TimingTest13 - ReadFromDisk timing test presented in BYTE, Fall, 1985, page 200. Uses the file created by first running TimingTest12.

65. TimingTest14 - Timing test consists of sorting a randomly filled 10000 element INTEGER array using F-Basic's INTSORT library routine. Due to limited space on the F-Basic system disk, the object file created by the compiler for this program should be directed to a different disk. For example, assuming again a two floppy drive system, one could invoke the compiler with:

```
l>FB dfl:TimingTest14 dfl:test.bin
```

This will produce the object file in a file named test.bin on the first external drive.

66. TimingTest15 - Timing test consists of sorting a randomly filled TEXT*5 array of 10000 elements using F-Basic's TXTSORT library function. Again, as with TimingTest14, the object file should be directed away from the F-Basic system disk. Additionally, the _0 option, as explained in Chapter 1, should be utilized to successfully compile the program, due to its large data storage requirements.
67. ZTest1 - This program accepts as input N and K, the respective beginning and ending elements of an array to be sorted by the Merge-Sort algorithm. It first sorts the array with the elements in reverse order, printing out the number of compares. It then performs 100 sorts (randomly mixing up the array each time) and prints the average number of compares made over the 100 trials.

NOTE: This program is not for the squeemish as direct manipulation of operating system memory and data structures is used to effect the visual displays. F-Basic's &SYSLIB interface is also used for direct ROM Kernel routine calls.

Direct Assembly Code is utilized by using F-Basic's INSERT statement.

Example Input: 100 and then 200

As with programs TimingTest14 and TimingTest15, the object file for this program should be directed away from the F-Basic's system disk.

68. ZTest2 - Program written by Neal Countryman to visually compare the performance of the Brute Force Search, Binary Search, and Modified Binary Search algorithms on an ordered list of 1793 elements. 1000 trials of the respective techniques are used on a randomly generated target, and the number of integer compares with array elements (and thus, the effective speed of the algorithms) are tabulated and displayed by the program in a unique way. Requires the presence of the file DSI.Screen on the F-Basic Sample Programs disk (assumed to be in dfl:!). This is an IFF file used by the program.

NOTE: This program is not for the squeemish as direct manipulation of operating system memory and data structures is used to effect the visual displays.

Upon execution, the program can be "started" by moving the mouse to the on/off switch in the upper right corner of the display and clicking on it. To start any of the three timing tests, the mouse should be moved to the respective start box at the bottom of the screen and clicked on the appropriate choice. To end the program, reposition the mouse to the on/off switch in the upper right corner and click on it.

The visual rendering of the Binary Search and Modified Binary Search are not as dynamic as that of the Brute Force Search algorithm, but illustrates how much better these algorithms are in terms of number of compares.

As with programs TimingTest14, TimingTest15, and ZTest1, the object file for this program should be directed away from the F-Basic system disk.

Additions:

GfxsTest28- Program uses F-Basic's COLOR_BOX, COLOR_FLOOD, and SINGLE_CLICK event to simulate a simple requestor in a custom window.

GfxsTest29- Program demonstrates the use of F-Basic's RECORD structures and ROM Kernel calls to the Graphics and DiskFont libraries to change the fonts displayed in a custom window.

GfxsTest30- Program demonstrates F-Basic's ability to interact with the right joystick/mouse port. Plug a regular joystick into the right port and this program will report the stick moves and button presses.

GfxsTest31- Program directly accesses the X and Y coordinates of the mouse pointer at any time, and reports them to the user 10 times.

GfxsTest32- Program demonstrates the SINGLE_CLICK event in F-Basic. Push the left mouse button down to mark the beginning of a line, and release the button to signify the end. The line is then drawn in a random color. After 15 lines, the program terminates.

Dear Friend:

Thank you for purchasing the F-Basic System. It is important that you fill out this card and return it to Delphi Noetic Systems, Inc., so that we can send you the latest technical information about our current and future products. Only a registered user will be afforded this opportunity, so please take a few moments now to complete the card.

F-Basic Serial Number DLM037
Date Purchased _____
Name _____
Address _____
(Please do not use a P.O. Box)
City _____ State _____ Zip _____ Country _____
Company Name (if applicable) _____
Home Phone # (____) - _____ Business Phone # (____) - _____

I have read and agree to the terms of the F-Basic Software License Agreement.

☐ Please include me in future promotional mailings from DNS, Inc.

☐ Please do not include me in future promotional mailings from DNS, Inc.

Signature

Date

Place
Stamp
Here

F-Basic System Registration
Delphi Noetic Systems, Inc.
P.O. Box 7722
Rapid City, SD 57709

Delphi Noetic Systems, Inc.

If your diskettes become damaged or worn out ...
Delphi Noetic Systems, Inc. realizes that your
original F-Basic System software diskettes may become
damaged or worn out through continued use. So we're
making it possible to exchange a damaged diskette for
a new one.

How to exchange a diskette ...

Send the defective diskette back to Delphi Noetic
Systems, Inc. with a check or money order for ten
dollars and we'll send you a replacement diskette by
return mail within 10 days.

**THIS POLICY APPLIES ONLY TO ORIGINAL F-BASIC SYSTEM
DISKETTES CONTAINING Delphi Noetic Systems, Inc.
SOFTWARE.**

**Return this form with your damaged diskette and check
or money order, payable to Delphi Noetic Systems, Inc.
to:**

Diskette Replacement
Delphi Noetic Systems, Inc.
P.O. Box 7722
2040 West Main - Suite 211
Rapid City, SD 57709

Diskette Exchange Form - Please print clearly

Owner's Name _____
Address _____
City _____ State _____ Zip _____
Phone _____

Delphi Noetic Systems, Inc.

=====

The F-Basic System

=====

The F-Basic System's System disk is bootable! Simply insert it in the drive in the place of your Workbench disk. Alternatively, invoke the F-Basic System from the CLI, as explained in Chapter 1 of the F-Basic System User's Manual. Please make a backup copy of the F-Basic System disks to use!

**The F-Basic System
System Disk**

Serial # DLM037
Delphi Noetic Systems, Inc.

**The F-Basic System
Sample Programs Disk**

Delphi Noetic Systems, Inc.